ppci Documentation

Release 0.5.9

Windel Bouwman

Sep 13, 2020

Contents

CHAPTER 1

Quickstart

1.1 Installation

1.1.1 Using pip

Install ppci in a virtualenv environment:

```
$ virtualenv sandbox
$ source sandbox/bin/activate
(sandbox) $ pip install ppci
(sandbox) $ ppci-build -h
```

If ppci installed correcly, you will get a help message of the *ppci-build* commandline tool.

1.1.2 Manually

Alternatively you can download a zip package from PyPI or from GitHub. Unpack the source archive and open a console in this directory.

```
$ virtualenv sandbox
$ source sandbox/bin/activate
(sandbox) $ git clone https://github.com/windelbouwman/ppci.git
(sandbox) $ cd ppci
(sandbox) $ python setup.py install
(sandbox) $ ppci-build -h
```

If ppci installed correcly, you will get a help message of the *ppci-build* commandline tool.

1.2 Compile some code!

Now lets compile some code via the *high level api functions*:

```
>>> import io
>>> from ppci.api import cc, get_arch
```

(continues on next page)

(continued from previous page)

```
>>> source = "int add(int a, int b) { return a + b; }"
>>> f = io.StringIO(source)
>>> obj = cc(f, get_arch('arm'))
>>> obj
CodeObject of 48 bytes
```

Let review what we have just done:

- We defined a simple add function in C
- We compiled this with the *ppci.api.cc()* function to arm object code

1.3 Example projects

The examples folder in the ppci sourcecode contains some demo projects that can be built using PPCI.

stm32f4 example

To build the blinky project do the following:

```
$ cd examples/blinky
$ ppci-build
```

Flash the hexfile using your flashtool of choice on the stm32f4discovery board and enjoy the magic.

arduino example

To build and flash the arduino blink led example, use the following commands:

```
$ cd examples/avr/arduino-blinky
$ ppci-build
$ avrdude -v -P /dev/ttyACM0 -c arduino -p m328p -U flash:w:blinky.hex
```

Linux x86_64 example

To build the hello world for 64-bit linux, go here:

```
$ cd examples/linux64/hello
$ ppci-build
$ ./hello
```

Or run the snake demo under linux:

```
$ cd examples/linux64/snake
$ ppci-build
$ ./snake
```

1.4 Next steps

If you have checked out the examples, head over to the *howto*, *api* and *reference* sections to learn more!

CHAPTER 2

Howto

This section contains several howto documents.

2.1 Creating a toy language

In this how-to, we will develop our own toy language. We will use textx to define our own language and use the ppci backend for optimization and code generation.

As an example we will create a simple language that can calculate simple expressions and use variables. An example of this toy language looks like this:

```
b = 2;
c = 5 + 5 * b;
d = 133 * c - b;
print b;
print c;
```

The language is very limited (which makes it easy to implement), but it contains enough for an example. The example above is stored in a file called 'example.tcf' (tcf stands for toy calculator format).

2.1.1 Part 0 - preparation

Before we can begin creating the toy language compiler, we need the required dependencies. For that a virtualenv can be created like this:

```
[windel@hoefnix toydsl]$ virtualenv dslenv
Using base prefix '/usr'
New python executable in /home/windel/HG/ppci/examples/toydsl/dslenv/bin/python3
Also creating executable in /home/windel/HG/ppci/examples/toydsl/dslenv/bin/python
Installing setuptools, pip, wheel...done.
[windel@hoefnix toydsl]$ source dslenv/bin/activate
(dslenv) [windel@hoefnix toydsl]$ pip install textx ppci
Collecting textx
Collecting ppci
  Using cached ppci-0.5-py3-none-any.whl
Collecting Arpeggio (from textx)
```

(continues on next page)

(continued from previous page)

```
Installing collected packages: Arpeggio, textx, ppci
Successfully installed Arpeggio-1.5 ppci-0.5 textx-1.4
(dslenv) [windel@hoefnix toydsl]$
```

After this step, we now have a virtual environment with textx and ppci installed.

2.1.2 Part 1 - textx

In this part the parsing of the language will be done. A great deal will be done by textx. For a detailed explanation of the workings of textx, please see: http://igordejanovic.net/textX/

Lets define a grammar file, called toy.tx:

```
Program: statements*=Statement;
Statement: (PrintStatement | AssignmentStatement) ';';
PrintStatement: 'print' var=ID;
AssignmentStatement: var=ID '=' expr=Expression;
Expression: Sum;
Sum: Product (('+'|'-') Product)*;
Product: Value ('*' Value)*;
Value: ID | INT | ('(' Expression ')');
```

This grammar is able to parse our toy language. Next we create a python script to load this grammar and parse the toy example program:

```
from textx.metamodel import metamodel_from_file

toy_mm = metamodel_from_file('toy.tx')

# Load the program:
program = toy_mm.model_from_file('example.tcf')

for statement in program.statements:
    print(statement)
```

Now if we run this file, we see the following:

```
(dslenv) [windel@hoefnix toydsl]$ python toy.py
<textx:AssignmentStatement object at 0x7f20c9d87cc0>
<textx:AssignmentStatement object at 0x7f20c9d87908>
<textx:AssignmentStatement object at 0x7f20c9d870b8>
<textx:PrintStatement object at 0x7f20c9d87ac8>
<textx:PrintStatement object at 0x7f20c9d95588>
```

We now have a simple parser for the toy language, and can parse it.

2.1.3 Part 2 - connecting the backend

Now that we can parse the dsl, it is time to create new code from the parsed format. To generate code, first the program must be translated to ir code.

The following snippet creates an IR-module, a procedure and a block to store instructions in. Instructions at this point are not machine instructions but abstract instructions that can be translated into any kind of machine code later on.

```
from ppci import ir
ir_module = ir.Module('toy')
ir_function = ir.Procedure('toy', ir.Binding.GLOBAL)
ir_module.add_function(ir_function)
```

(continues on next page)

(continued from previous page)

```
ir_block = ir.Block('entry')
ir_function.entry = ir_block
ir_function.add_block(ir_block)
```

Next, we need to translate each statement into some code, but we will do that later.

```
for statement in program.statements:
    print(statement)
```

First we will add the closing code, that verifies our own constructed module, and compiles the ir code to object code, links this and creates an oj file.

```
ir_block.add_instruction(ir.Exit())
```

The code above creates an Exit instruction and adds the instruction to the block. Next we can verify the IR-code, to make sure that the program we created contains no errors. The ir_to_object function translates the program from IR-code into an object for the given target architecture, in this case x86_64, but you could as well use AVR or riscv here.

```
from ppci.irutils import Verifier
from ppci import api
Verifier().verify(ir_module)
obj1 = api.ir_to_object([ir_module], 'x86_64')
obj = api.link([obj1])
print(obj)
```

The printed object shows that it conains 11 bytes.

```
(dslenv) [windel@hoefnix toydsl]$ python toy.py
...
CodeObject of 11 bytes
(dslenv) [windel@hoefnix toydsl]$
```

We can write the object to file using the following code:

```
with open('example.oj', 'w') as f:
    obj.save(f)
```

The oj file is a ppci format for object files, pronounced 'ojee'. It is a readable json format with the object information in it:

```
{
  "arch": "x86_64",
  "images": [],
  "relocations": [
    {
      "offset": "0x4",
      "section": "code",
      "symbol": "toy_toy_epilog",
      "type": "apply_b_jmp32"
    }
  ],
  "sections": [
    {
      "address": "0x0",
      "alignment": "0x4",
      "data": "",
      "name": "data"
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "address": "0x0",
      "alignment": "0x4",
      "data": "55488bece9000000005dc3",
      "name": "code"
    }
  ],
  "symbols": [
    {
      "name": "toy_toy",
      "section": "code",
      "value": "0x0"
    },
    {
      "name": "toy_toy_block_entry",
      "section": "code",
      "value": "0x4"
    },
    {
      "name": "toy_toy_epilog",
      "section": "code",
      "value": "0x9"
    }
  ]
}
```

As you can see, there are two sections, for code and for data. The code section contains some bytes. This is x86_64 machine code.

2.1.4 Part 3 - translating the elements

In this part we will create code snippets for each type of TCF code. For this we will use the textx context processor system, and we will also rewrite the initial code such that we have a class that can translate TCF code into IR-code. The entry point to the class will be a compile member function that translates a TCF file into a IR-module.

The whole script now looks like this:

```
import logging
import struct
from textx.metamodel import metamodel_from_file
from ppci import ir
from ppci.irutils import verify_module
from ppci import api


def pack_string(txt):
    ln = struct.pack('<Q', len(txt))
    return ln + txt.encode('ascii')


class TcfCompiler:
    """ Compiler for the Tcf language """
    logger = logging.getLogger('tcfcompiler')

    def __init__(self):
        self.int_size = 8
        self.int_type = ir.i64
        self.toy_mm = metamodel_from_file('toy.tx')
        self.toy_mm.register_obj_processors({
            'PrintStatement': self.handle_print,
            'AssignmentStatement': self.handle_assignment,
```

(continues on next page)

(continued from previous page)

```
            'Expression': self.handle_expression,
            'Sum': self.handle_sum,
            'Product': self.handle_product,
            })

    def compile(self, filename):
        self.variables = {}

        # Prepare the module:
        ir_module = ir.Module('toy')
        self.io_print2 = ir.ExternalProcedure('io_print2', [ir.ptr, self.int_type])
        ir_module.add_external(self.io_print2)
        ir_function = ir.Procedure('toy_toy', ir.Binding.GLOBAL)
        ir_module.add_function(ir_function)
        self.ir_block = ir.Block('entry')
        ir_function.entry = self.ir_block
        ir_function.add_block(self.ir_block)

        # Load the program:
        self.toy_mm.model_from_file('example.tcf')

        # Close the procedure:
        self.emit(ir.Exit())

        verify_module(ir_module)
        return ir_module

    def emit(self, instruction):
        self.ir_block.add_instruction(instruction)
        return instruction

    def handle_print(self, print_statement):
        self.logger.debug('print statement %s', print_statement.var)
        name = print_statement.var
        value = self.load_var(name)
        label_data = pack_string('{} :'.format(name))
        label = self.emit(ir.LiteralData(label_data, 'label'))
        label_ptr = self.emit(ir.AddressOf(label, 'label_ptr'))
        self.emit(ir.ProcedureCall(self.io_print2, [label_ptr, value]))

    def handle_assignment(self, assignment):
        self.logger.debug(
            'assign %s = %s', assignment.var, assignment.expr)
        name = assignment.var
        assert isinstance(name, str)

        # Create the variable on stack, if not already present:
        if name not in self.variables:
            alloc = self.emit(
                ir.Alloc(name + '_alloc', self.int_size, self.int_size))
            addr = self.emit(ir.AddressOf(alloc, name + '_addr'))
            self.variables[name] = addr
        mem_loc = self.variables[name]
        value = assignment.expr.ir_value
        self.emit(ir.Store(value, mem_loc))

    def handle_expression(self, expr):
        self.logger.debug('expression')
        expr.ir_value = expr.val.ir_value

    def handle_sum(self, sum):
```

(continues on next page)

(continued from previous page)

```
        """ Process a sum element """
        self.logger.debug('sum')
        lhs = sum.base.ir_value
        for term in sum.terms:
            op = term.operator
            rhs = term.value.ir_value
            lhs = self.emit(ir.Binop(lhs, op, rhs, 'sum', self.int_type))
        sum.ir_value = lhs

    def handle_product(self, product):
        self.logger.debug('product')
        lhs = self.get_value(product.base)
        for factor in product.factors:
            rhs = self.get_value(factor.value)
            lhs = self.emit(ir.Binop(lhs, '*', rhs, 'prod', self.int_type))
        product.ir_value = lhs

    def get_value(self, value):
        if isinstance(value, int):
            ir_value = self.emit(ir.Const(value, 'constant', self.int_type))
        elif isinstance(value, str):
            ir_value = self.load_var(value)
        else:  # It must be an expression!
            ir_value = value.ir_value
        return ir_value

    def load_var(self, var_name):
        mem_loc = self.variables[var_name]
        return self.emit(ir.Load(mem_loc, var_name, self.int_type))


tcf_compiler = TcfCompiler()
ir_module = tcf_compiler.compile('example.tcf')

obj1 = api.ir_to_object([ir_module], 'x86_64')
obj2 = api.c3c(['bsp.c3', '../../librt/io.c3'], [], 'x86_64')
obj3 = api.asm('linux.asm', 'x86_64')
obj = api.link([obj1, obj2, obj3], layout='layout.mmap')

print(obj)
with open('example.oj', 'w') as f:
    obj.save(f)

# Create a linux elf file:
api.objcopy(obj, 'code', 'elf', 'example')
```

And the textx description is modified to include sum and product terms:

```
Program: statements*=Statement;
Statement: (PrintStatement | AssignmentStatement) ';';
PrintStatement: 'print' var=ID;
AssignmentStatement: var=ID '=' expr=Expression;
Expression: val=Sum;
Sum: base=Product terms*=ExtraTerm;
ExtraTerm: operator=Operator value=Product;
Operator: '+' | '-';
Product: base=Value factors*=ExtraFactor;
ExtraFactor: operator='*' value=Value;
Value: ID | INT | ('(' Expression ')');
```

When we run this script, the output is the following:

```
(dslenv) [windel@hoefnix toydsl]$ python toy.py
CodeObject of 117 bytes
(dslenv) [windel@hoefnix toydsl]$
```

As we can see, the object file has increased in size because we translated the elements.

2.1.5 Part 4 - Creating a linux executable

In this part we will create a linux executable from the object code we created. We will do this very low level, without libc, directly using the linux syscall api.

We will start with the low level assembly glue code (linux.asm):

```
section reset

start:
    call toy_toy
    call bsp_exit

bsp_syscall:
    mov rax, rdi ; abi param 1
    mov rdi, rsi ; abi param 2
    mov rsi, rdx ; abi param 3
    mov rdx, rcx ; abi param 4
    syscall
    ret
```

In this assembly snippet, we defined a sequence of code in the reset section which calls our toy_toy function and next the bsp_exit function. Bsp is an abbreviation for board support package, and we need it to connect other code to the platform we run on. The syscall assembly function calls the linux kernel with four parameters.

Next we define the rest of the bsp in bsp.c3:

```
module bsp;

public function void putc(byte c)
{
  syscall(1, 1, cast<int64_t>(&c), 1);
}

function void exit()
{
    syscall(60, 0, 0, 0);
}

function void syscall(int64_t nr, int64_t a, int64_t b, int64_t c);
```

Here we implement two syscalls, namely putc and exit.

For the print function, we will refer to the already existing io module located in the librt folder of ppci. To compile and link the different parts we use the following snippet:

```
obj1 = api.ir_to_object([ir_module], 'x86_64')
obj2 = api.c3c(['bsp.c3', '../../librt/io.c3'], [], 'x86_64')
obj3 = api.asm('linux.asm', 'x86_64')
obj = api.link([obj1, obj2, obj3], layout='layout.mmap')
```

In this snippet, three object files are created. obj1 contains our toy languaged compiled into x86 code. obj2 contains the c3 bsp and io code. obj3 contains the assembly sourcecode.

For the link command we also use a layout file, telling the linker where it must place which piece of the object file. In the case of linux, we use the following (layout.mmap):

```
MEMORY code LOCATION=0x40000 SIZE=0x10000 {
    SECTION(reset)
    ALIGN(4)
    SECTION(code)
}

MEMORY ram LOCATION=0x20000000 SIZE=0xA000 {
    SECTION(data)
}
```

As a final step, we invoke the objcopy command to create a linux ELF executable:

```
# Create a linux elf file:
api.objcopy(obj, 'code', 'elf', 'example')
```

This command creates a file called 'example', which is an ELF file for linux. The file can be inspected with objdump:

```
(dslenv) [windel@hoefnix toydsl]$ objdump example -d

example:     file format elf64-x86-64


Disassembly of section code:

000000000004001c <toy_toy>:
   4001c:   55                      push   %rbp
   4001d:   41 56                   push   %r14
   4001f:   41 57                   push   %r15
   40021:   48 81 ec 18 00 00 00    sub    $0x18,%rsp
   40028:   48 8b ec                mov    %rsp,%rbp

000000000004002b <toy_toy_block_entry>:
   4002b:   49 be 02 00 00 00 00    movabs $0x2,%r14
   40032:   00 00 00
   40035:   4c 89 75 00             mov    %r14,0x0(%rbp)
   40039:   4c 8b 7d 00             mov    0x0(%rbp),%r15
   4003d:   49 be 05 00 00 00 00    movabs $0x5,%r14

...
```

We can now run the executable:

```
(dslenv) [windel@hoefnix toydsl]$ ./example
Segmentation fault (core dumped)
(dslenv) [windel@hoefnix toydsl]$
```

Sadly, this is not exactly what we hoped for!

The problem here is that we did not call the io_print function with the proper arguments. To fix this, we can change the print handling routine like this:

```
def handle_print(self, print_statement):
    self.logger.debug('print statement %s', print_statement.var)
    name = print_statement.var
    value = self.load_var(name)
    label_data = pack_string('{} :'.format(name))
    label = self.emit(ir.LiteralData(label_data, 'label'))
    self.emit(ir.ProcedureCall('io_print2', [label, value]))
```

We use here io_print2, which takes a label and a value. The label must be packed as a pascal style string, meaning a length integer followed by the string data. We can implement this string encoding with the following function:

```
def pack_string(txt):
    ln = struct.pack('<Q', len(txt))
    return ln + txt.encode('ascii')
```

Now we can compile the TCF file again, and check the result:

```
(dslenv) [windel@hoefnix toydsl]$ python toy.py
CodeObject of 1049 bytes
(dslenv) [windel@hoefnix toydsl]$ ./example
b :0x00000002
c :0x0000000F
(dslenv) [windel@hoefnix toydsl]$ cat example.tcf
b = 2;
c = 5 + 5 * b;
d = 133 * c - b;
print b;
print c;
(dslenv) [windel@hoefnix toydsl]$
```

As we can see, the compiler worked out correctly!

2.1.6 Final words

In this tutorial we have seen how to create a simple language. The entire example for this code can be found in the examples/toydsl directory in the ppci repository.

2.2 How to write a new backend

This section describes how to add a new backend. The best thing to do is to take a look at existing backends, like the backends for ARM and X86_64.

A backend consists of the following parts:

1. Register descriptions
2. Instruction descriptions
3. Template descriptions
4. Function calling machinery
5. Architecture description

2.2.1 Register description

A backend must describe what kinds of registers are available. To do this define for each register class a subclass of `ppci.arch.isa.Register`.

There may be several register classes, for example 8-bit and 32-bit registers. It is also possible that these classes overlap.

```
from ppci.arch.encoding import Register

class X86Register(Register):
    bitsize = 32

class LowX86Register(Register):
    bitsize = 8
```

(continues on next page)

(continued from previous page)

```
AL = LowX86Register('al', num=0)
AH = LowX86Register('ah', num=4)
EAX = X86Register('eax', num=0, aliases=(AL, AH))
```

2.2.2 Tokens

Tokens are the basic building blocks of complete instructions. They correspond to byte sequences of parts of instructions. Good examples are the opcode token of typically one byte, the prefix token and the immediate tokens which optionally follow an opcode. Typically RISC machines will have instructions with one token, and CISC machines will have instructions consisting of multiple tokens.

To define a token, subclass the `ppci.arch.token.Token` and optionally add bitfields:

```
from ppci.arch.token import Token, bit_range

class Stm8Token(Token):
    class Info:
        size = 8

    opcode = bit_range(0, 8)
```

In this example an 8-bit token is defined with one field called 'opcode' of 8 bits.

2.2.3 Instruction description

An important part of the backend is the definition of instructions. Every instruction for a specific machine derives from *ppci.arch.encoding.Instruction*.

Lets take the `nop` example of stm8. This instruction can be defined like this:

```
from ppci.arch.encoding import Instruction, Syntax

class Nop(Instruction):
    syntax = Syntax(['nop'])
    tokens = [Stm8Token]
    patterns = {'opcode': 0x9d}
```

Here the "nop" instruction is defined. It has a syntax of `nop`. The syntax is used for creating a nice string representation of the object, but also during parsing of assembly code. The tokens contains a list of what tokens this instruction contains.

The `patterns` attribute contains a list of bitfield patterns. In this case the `opcode` field is set to the fixed pattern 0x9d.

Instructions are also usable directly, like this:

```
>>> ins = Nop()
>>> str(ins)
'nop'
>>> ins
<Nop object at ...>
>>> type(ins)
<class 'Nop'>
>>> ins.encode()
b'\x9d'
```

Often, an instruction does not have a fixed syntax. Often an argument can be specified, for example the stm8 `adc` instruction:

```
from ppci.arch.encoding import Operand

class Stm8ByteToken(Token):
    class Info:
        size = 8

    byte = bit_range(0, 8)

class AdcByte(Instruction):
    imm = Operand('imm', int)
    syntax = Syntax(['adc', ' ', 'a', ',', ' ', imm])
    tokens = [Stm8Token, Stm8ByteToken]
    patterns = {'opcode': 0xa9, 'byte': imm}
```

The `imm` attribute now functions as a variable instruction part. When constructing the instruction, it must be given as an argument:

```
>>> ins = AdcByte(0x23)
>>> str(ins)
'adc a, 35'
>>> type(ins)
<class 'AdcByte'>
>>> ins.encode()
b'\xa9#'
>>> ins.imm
35
```

As a benefit of specifying syntax and patterns, the default decode classmethod can be used to create an instruction from bytes:

```
>>> ins = AdcByte.decode(bytes([0xa9,0x10]))
>>> ins
<AdcByte object at ...>
>>> str(ins)
'adc a, 16'
```

Another option of constructing instruction classes is adding different instruction classes to each other:

```
from ppci.arch.encoding import Operand

class Sbc(Instruction):
    syntax = Syntax(['sbc', ' ', 'a'])
    tokens = [Stm8Token]
    patterns = {'opcode': 0xa2}

class Byte(Instruction):
    imm = Operand('imm', int)
    syntax = Syntax([',', ' ', imm])
    tokens = [Stm8ByteToken]
    patterns = {'byte': imm}

SbcByte = Sbc + Byte
```

In the above example, two instruction classes are defined. When combined, the tokens, syntax and patterns are combined into the new instruction:

```
>>> ins = SbcByte.decode(bytes([0xa2,0x10]))
>>> str(ins)
'sbc a, 16'
>>> type(ins)
<class 'ppci.arch.encoding.SbcByte'>
```

2.2.4 Relocations

Most instructions can be encoded directly, but some refer to a label which is not known at the time a separate instruction is created. The answer to this problem is relocation information. When generating instructions also relocation information is emitted. During link time, or during loading, the relocations are resolved and the instructions are patched.

To define a relocation, subclass `ppci.arch.encoding.Relocation`.

```
from ppci.arch.encoding import Relocation

class Stm8WordToken(Token):
    class Info:
        size = 16
        endianness = 'big'

    word = bit_range(0, 16)

class Stm8Abs16Relocation(Relocation):
    name = 'abs16'
    token = Stm8WordToken
    field = 'word'

    def calc(self, symbol_value, reloc_value):
        return symbol_value
```

To use this relocation, use it in instruction's `relocations` function:

```
class Jp(Instruction):
    label = Operand('label', str)
    syntax = Syntax(['jp', ' ', label])
    tokens = [Stm8Token, Stm8WordToken]
    patterns = {'opcode': 0xcc}

    def relocations(self):
        return [Stm8Abs16Relocation(self.label, offset=1)]
```

The relocations function returns a list of relocations for this instruction. In this case it is one relocation entry at offset 1 into the instruction.

2.2.5 Instruction groups

Instructions often not come one by one. They are usually grouped into a set of instructions, or an instruction set architecture (ISA). An isa can be created and instructions can be added to it, like this:

```
from ppci.arch.isa import Isa
my_isa = Isa()
my_isa.add_instruction(Nop)
```

The instructions of an isa can be inspected:

```
>>> my_isa.instructions
[<class 'Nop'>]
```

Instead of adding each instruction manually to an isa, one can also specify the isa in the class definition of the instruction:

```
class Stm8Instruction(Instruction):
    isa = my_isa
```

The class Stm8Instruction and all of its subclasses will now be automatically added to the isa.

Often there are some common instructions for data definition, such as the `db` instruction to define a byte. These are already defined in `data_instructions`. Isa's can be added to each other to combine them, like this:

```
from ppci.arch.data_instructions import data_isa
my_complete_isa = my_isa + data_isa
```

2.2.6 Instruction selection patterns

In order for the compiler to know what instructions must be used when, use can be made of the built-in pattern matching for instruction selection. To do this, specify a series of patterns with a possible implementation for the backend.

```
@my_isa.pattern('a', 'ADDU8(a, CONSTU8)', size=2, cycles=3, energy=2)
def pattern_const(context, tree, c0):
    value = tree[1].value
    context.emit(AdcByte(value))
    return A
```

In the function above a function is defined that matches the pattern for adding a constant to the accumulator (a) register. The instruction selector will use the information about size, cycles and energy to determine the best choice depending on codegeneration options given. For example, if the compiler is run with option to optimize for size, the size argument will be weighted heavier in the determination of the choice of pattern.

When a pattern is selected, the function is run, and the corresponding instruction must be emitted into the context which is given to the function as a first argument.

See also: *ppci.arch.isa.Isa.pattern()*.

Note: this example uses an accumulator machine, a better example could be given using a register machine.

2.2.7 Architecture description

Now that we have some instructions defined, it is time to include them into a target architecture. To create a target architecture, subclass *ppci.arch.arch.Architecture*.

A subclass must implement a fair amount of member functions. Lets examine them one by one.

Code generating functions

There are several functions that are expected to generate code. Code can be generated by implementing these functions as Python generators, but returning a list of instructions is also possible. All these functions names start with `gen_`.

These functions are for prologue / epilogue:

- *ppci.arch.arch.Architecture.gen_prologue()*
- *ppci.arch.arch.Architecture.gen_epilogue()*

For creating a call:

- *ppci.arch.arch.Architecture.gen_call()*

During instruction selection phase, the `gen_call` function is called to generate code for function calls.

The member functions *ppci.arch.arch.Architecture.gen_prologue()* and *ppci.arch.arch.Architecture.gen_epilogue()* are called at the very end stage of code generation of a single function.

Architecture information

Most frontends also need some information, but not all about the target architecture. For this create architecture info object using *ppci.arch.arch_info.ArchInfo*. This class holds information about basic type sizes, alignment and endianness of the architecture.

2.3 How to write an optimizer

This section will dive into the pecularities on how to implement an optimizer scheme. The optimizer will be an IR optimizer, in the sense that is transforms IR-code into new (and improved) IR-code. This makes the optimizer both programming language and target architecture independent.

2.3.1 The optimization

The optimizer we will implement in this example is an optimization that deletes redundant branching. For example, in the following code, branch *b* is never taken:

```
module optimizable;

function i32 add(i32 z) {
  entry: {
    i32 x = 1;
    i32 y = 2;
    cjmp x < y ? a : b;
  }
  a: {
    jmp c;
  }
  b: {
    jmp c;
  }
  c: {
    return x;
  }
}
```

It can be easily seen that *b* is never jumped to, because $x < y$ is always true. Time to write an optimization that simplifies this!

2.3.2 The implementation

To implement an optimalisation, the *ppci.opt.transform.ModulePass* must be subclassed. For this example, *ppci.opt.transform.InstructionPass* will be used.

```
import operator
from ppci.opt.transform import InstructionPass
from ppci import ir

class SimpleComparePass(InstructionPass):
    def on_instruction(self, instruction):
        if isinstance(instruction, ir.CJump) and \
                isinstance(instruction.a, ir.Const) and \
                isinstance(instruction.b, ir.Const):
            a = instruction.a.value
            b = instruction.b.value
            mp = {
                '==': operator.eq,
```

(continues on next page)

(continued from previous page)

```
                '<': operator.lt, '>': operator.gt,
                '>=': operator.ge, '<=': operator.le,
                '!=': operator.ne
                }
            if mp[instruction.cond](a, b):
                label = instruction.lab_yes
            else:
                label = instruction.lab_no
            block = instruction.block
            block.remove_instruction(instruction)
            block.add_instruction(ir.Jump(label))
            instruction.delete()
```

The implementation first checks if the instruction is a conditional jump and if both inputs are constant. Then the constants are compared using the operator module. Finally a *ppci.ir.Jump* instruction is created. This instruction is added to the block after the *ppci.ir.CJump* instruction is removed.

First load the IR-module from file. To do this, first create an in memory file with io.StringIO. Then load this file with `ppci.irutils.Reader`.

```
>>> import io
>>> f = io.StringIO("""
... module optimizable;
... global function i32 add(i32 z) {
...   entry: {
...     i32 x = 1;
...     i32 y = 2;
...     cjmp x < y ? a : b;
...   }
...   a: {
...     jmp c;
...   }
...   b: {
...     jmp c;
...   }
...   c: {
...     return x;
...   }
... }
... """)
>>> from ppci import irutils
>>> mod = irutils.Reader().read(f)
>>> print(mod)
module optimizable
```

Now run the optimizer pass:

```
>>> opt_pass = SimpleComparePass()
>>> opt_pass.run(mod)
```

Next delete all unreachable blocks to make sure the module is valid again:

```
>>> mod.functions[0].delete_unreachable()
```

Now print the optimized module:

```
>>> f2 = io.StringIO()
>>> irutils.Writer(f2).write(mod)
>>> print(f2.getvalue())
module optimizable;

```

(continues on next page)

(continued from previous page)

```
global function i32 add(i32 z) {
  entry: {
    i32 x = 1;
    i32 y = 2;
    jmp a;
  }

  a: {
    jmp c;
  }

  c: {
    return x;
  }

}
```

This optimization is implemented in *ppci.opt.cjmp.CJumpPass*.

2.4 JITting

Warning: This section is a work in progress. It is outlined as to how things should work, but it is not thorougly tested. Also, keep in mind that C support is very premature. An alternative is C3.

This howto is about how to JIT (just-in-time-compile) code and use it from Python. It can occur that at some point in time, you have some Python code that becomes a performance bottleneck. At this point, you have multiple options:

- Rewrite your code in C/Fortran/Rust/Go/Swift, compile it to machine code with GCC or similar compiler and load it with SWIG/ctypes.
- Use PyPy, which contains a built-in JIT functionality. Usually the usage of PyPy means more speed.
- Use a specialized JIT engine, like Numba.

In this HowTo we will implement our own specialized JIT engine, using PPCI as a backend.

To do this, first we need some example code. Take the following function as an example:

```
def x(a, b):
    return a + b + 13
```

This function does some magic calculations :)

```
>>> x(2, 3)
18
```

2.4.1 C-way

Now, after profiling we could potentially discover that this function is a bottleneck. We may decide to rewrite it in C:

```
int x(int a, int b)
{
  return a + b + 13;
}
```

Having this function, we put this function in a Python string and compile it.

```
>>> from ppci import api
>>> import io
>>> src = io.StringIO("""
... int x(int a, int b) {
...   return a + b + 13;
... }""")
>>> arch = api.get_current_arch()
>>> obj = api.cc(src, arch, debug=True)
>>> obj
CodeObject of ... bytes
```

Now that the object is compiled, we can load it into the current Python process:

```
>>> from ppci.utils.codepage import load_obj
>>> m = load_obj(obj)
>>> dir(m)
[..., 'x']
>>> m.x
<CFunctionType object at ...>
```

Now, lets call the function:

```
>>> m.x(2, 3)
18
```

2.4.2 Python-way

Instead of translating our code to C, we can as well compile Python code directly, by using type hints and a restricted subset of the Python language. For this we can use the *ppci.lang.python* module:

```
>>> from ppci.lang.python import load_py
>>> f = io.StringIO("""
... def x(a: int, b: int) -> int:
...     return a + b + 13
... """)
>>> n = load_py(f)
>>> n.x(2, 3)
18
```

By doing this, we do not need to reimplement the function in C, but only need to add some type hints to make it work. This might be more preferable to C. Please note that integer arithmetic is arbitrary-precision in Python, but witth the compiled code above, large value will silently wrap around.

To easily compile some of your Python functions to native code, use the *ppci.lang.python.jit()* decorator:

```
from ppci.lang.python import jit

@jit
def y(a: int, b: int) -> int:
    return a + b + 13
```

Now the function can be called as a normal function, JIT compilation and calling native code is handled transparently:

```
>>> y(2, 3)
18
```

2.4.3 Calling Python functions from native code

In order to callback Python functions, we can do the following:

```
>>> def callback_func(x: int) -> None:
...     print('x=', x)
...
>>> f = io.StringIO("""
... def x(a: int, b: int) -> int:
...     func(a+3)
...     return a + b + 13
... """)
>>> o = load_py(f, imports={'func': callback_func})
>>> o.x(2, 3)
x= 5
18
```

2.4.4 Benchmarking and call overheads

To conclude this section, let's benchmark the original function `x` with which we started this section, and its JIT counterpart:

```
>>> import timeit
>>> timeit.timeit('x(2,3)', number=100000, globals={'x': x})
0.015114138000171806
>>> timeit.timeit('x(2,3)', number=100000, globals={'x': m.x})
0.07410199400010242
```

Turns out that the compiled code is actually slower. This is due to the fact that for a trivial function like that, argument conversion and call preparation overheads dominate the execution time. To see benefits of native code execution, we would need to JIT functions which perform many operations in a loop, e.g. while processing large arrays.

> **Warning:** Before optimizing anything, run a profiler. Your expectations about performance bottlenecks might be wrong!

2.5 Dealing with webassembly

In this tutorial we will see the possible ways to use web assembly.

2.5.1 Compiling wasm to native code

The first possible usage is to take a wasm module and compile it to native code. The idea is to take wasm code and compile it to native code.

First lets create some wasm code by using wasmfiddle:

https://wasdk.github.io/WasmFiddle/

```
int main() {
  return 42;
}
```

The wasm output of this is:

```
(module
  (table 0 funcref)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (result i32)
    (i32.const 42)
  )
)
```

Download this wasm file from wasm fiddle to your local drive. Now you can compile it to for example native riscv code:

```
$ python -m ppci.cli.wasmcompile -v program.wasm -m riscv -O 2 -S
$ cat f.out
    .section data
    .section code
main:
    sw x1, -8(x2)
    sw x8, -12(x2)
    mv x8, x2
    addi x2, x2, -12
    addi x2, x2, 0
block1:
    addi x10, x0, 42
    j main_epilog
main_epilog:
    addi x2, x2, 0
    addi x2, x2, 12
    lw x8, -12(x2)
    lw x1, -8(x2)
    jalr x0,x1, 0
    .align 4
```

In this example we compiled C code with one compiler to wasm and took this wasm and compiled it to riscv code using ppci.

Please see *WebAssembly* for the python api for using webassembly.

2.6 Code instrumentation

This is a howto on code instrumentation. Code instrumentation is the action of adding extra code to your program. A good example is function call tracing. With function call tracing, you can execute a custom action whenever a function is entered. It is also fun and easy to enter infinite recursions by doing so.

Lets demonstrate how this works with an example!

Say, we have a simple function, and would like to instrument this code. So, first define a function in C, called `my_add`, and turn it into IR-code:

```
>>> import io
>>> from ppci import api
>>> source = """
... int my_add(int a, int b) { return a + b; }
... """
>>> arch = api.get_arch('arm')
>>> module = api.c_to_ir(io.StringIO(source), arch)
>>> api.optimize(module, level=2)
>>> module.display()
module main;
```

(continues on next page)

(continued from previous page)

```
global function i32 my_add(i32 a, i32 b) {
  my_add_block0: {
    i32 tmp_3 = a + b;
    return tmp_3;
  }

}
```

Now comes the cool part, the addition of tracing functionality. Since we have IR-code, we can add tracing to it. This means the tracing functionality is target independent!

```
>>> from ppci.irutils.instrument import add_tracer
>>> add_tracer(module)
>>> module.display()
module main;

external procedure trace(ptr);

global function i32 my_add(i32 a, i32 b) {
  my_add_block0: {
    blob<7:1> func_name = literal '6d795f61646400';
    ptr name_ptr = &func_name;
    call trace(name_ptr);
    i32 tmp_3 = a + b;
    return tmp_3;
  }

}
```

Notice the extra code inserted! Now, we could turn this into machine code like this:

```
>>> print(api.ir_to_assembly([module], arch))
       section data
       global trace
       type trace func
       section data
       section code
       global my_add
       type my_add func
 my_add:
       push LR, R11
       mov R11, SP
       push R5, R6
       mov R6, R1
       mov R5, R2
 my_add_block0:
       ldr R1, my_add_literal_1
       bl trace
       add R0, R6, R5
       b my_add_epilog
 my_add_epilog:
       pop R5, R6
       pop PC, R11
       ALIGN(4)
 my_add_literal_0:
       db 109
       db 121
       db 95
       db 97
       db 100
```

(continues on next page)

(continued from previous page)

```
       db 100
       db 0
       ALIGN(4)
my_add_literal_1:
       dcd =my_add_literal_0
       ALIGN(4)
```

Notice here as well the extra call to the `trace` function.

CHAPTER 3

Reference

3.1 Api

The ppci library provides provides an intuitive api to the compiler, assembler and other tools. For example to assemble, compile, link and objcopy the msp430 blinky example project, the api can be used as follows:

```
>>> from ppci.api import asm, c3c, link, objcopy
>>> march = "msp430"
>>> o1 = asm('examples/msp430/blinky/boot.asm', march)
>>> o2 = c3c(['examples/msp430/blinky/blinky.c3'], [], march)
>>> o3 = link([o2, o1], layout='examples/msp430/blinky/msp430.mmap')
>>> objcopy(o3, 'flash', 'hex', 'blinky_msp430.hex')
```

Instead of using the api, a set of *commandline tools* are also prodived.

The api module contains a set of handy functions to invoke compilation, linking and assembling.

ppci.api.**asm**(*source*, *march*, *debug=False*)

Assemble the given source for machine march.

Parameters

- **source** (*str*) – can be a filename or a file like object.
- **march** (*str*) – march can be a *ppci.arch.arch.Architecture* instance or a string indicating the machine architecture.
- **debug** – generate debugging information

Returns A *ppci.binutils.objectfile.ObjectFile* object

```
>>> import io
>>> from ppci.api import asm
>>> source_file = io.StringIO("db 0x77")
>>> obj = asm(source_file, 'arm')
>>> print(obj)
CodeObject of 1 bytes
```

ppci.api.**archive**(*objs*)

Create an archive from multiple object files.

ppci.api.**c3c**(*sources*, *includes*, *march*, *opt_level=0*, *reporter=None*, *debug=False*, *outstream=None*)

Compile a set of sources into binary format for the given target.

Parameters

- **sources** – a collection of sources that will be compiled.
- **includes** – a collection of sources that will be used for type and function information.
- **march** – the architecture for which to compile.
- **reporter** – reporter to write compilation report to
- **debug** – include debugging information

Returns An object file

```
>>> import io
>>> from ppci.api import c3c
>>> source_file = io.StringIO("module main; var int a;")
>>> obj = c3c([source_file], [], 'arm')
>>> print(obj)
CodeObject of 4 bytes
```

ppci.api.**cc**(*source: io.TextIOBase*, *march*, *coptions=None*, *opt_level=0*, *debug=False*, *reporter=None*)

C compiler. compiles a single source file into an object file.

Parameters

- **source** – file like object from which text can be read
- **march** – The architecture for which to compile
- **coptions** – options for the C frontend
- **debug** – Create debug info when set to True

Returns an object file

```
>>> import io
>>> from ppci.api import cc
>>> source_file = io.StringIO("void main() { int a; }")
>>> obj = cc(source_file, 'x86_64')
>>> print(obj)
CodeObject of 20 bytes
```

ppci.api.**link**(*objects*, *layout=None*, *use_runtime=False*, *partial_link=False*, *reporter=None*, *debug=False*, *extra_symbols=None*, *libraries=None*, *entry=None*)

Links the iterable of objects into one using the given layout.

Parameters

- **objects** – a collection of objects to be linked together.
- **layout** – optional memory layout.
- **use_runtime** (*bool*) – also link compiler runtime functions
- **partial_link** – Set this to true if you want to perform a partial link. This means, undefined symbols are no error.
- **debug** (*bool*) – when true, keep debug information. Otherwise remove this debug information from the result.
- **extra_symbols** – a dict of extra symbols which can be used during linking.
- **libraries** – a list of libraries to use when searching for symbols.
- **entry** – the entry symbol where execution should begin.

Returns The linked object file

```
>>> import io
>>> from ppci.api import asm, c3c, link
>>> asm_source = io.StringIO("db 0x77")
>>> obj1 = asm(asm_source, 'arm')
>>> c3_source = io.StringIO("module main; var int a;")
>>> obj2 = c3c([c3_source], [], 'arm')
>>> obj = link([obj1, obj2])
>>> print(obj)
CodeObject of 8 bytes
```

ppci.api.**objcopy**(*obj: ppci.binutils.objectfile.ObjectFile*, *image_name: str*, *fmt: str*, *output_filename*)
Copy some parts of an object file to an output

ppci.api.**bfcompile**(*source*, *target*, *reporter=None*)
Compile brainfuck source into binary format for the given target

Parameters

- **source** – a filename or a file like object.
- **march** – a architecture instance or a string indicating the target.

Returns A new object.

```
>>> import io
>>> from ppci.api import bfcompile
>>> source_file = io.StringIO(">>[-]<<[->>+<<]")
>>> obj = bfcompile(source_file, 'arm')
>>> print(obj)
CodeObject of ... bytes
```

ppci.api.**construct**(*buildfile*, *targets=()*)
Construct the given buildfile.

Raise task error if something goes wrong.

ppci.api.**optimize**(*ir_module*, *level=0*, *reporter=None*)
Run a bag of tricks against the *ir-code*.

This is an in-place operation!

Parameters

- **ir_module** (ppci.ir.Module) – The ir module to optimize.
- **level** – The optimization level, 0 is default. Can be 0,1,2 or s 0: No optimization 1: some optimization 2: more optimization s: optimize for size
- **reporter** – Report detailed log to this reporter

ppci.api.**preprocess**(*f*, *output_file*, *coptions=None*)
Pre-process a file into the other file.

ppci.api.**get_arch**(*arch*)
Try to return an architecture instance.

Parameters **arch** – can be a string in the form of arch:option1:option2

```
>>> from ppci.api import get_arch
>>> arch = get_arch('msp430')
>>> arch
msp430-arch
>>> type(arch)
<class 'ppci.arch.msp430.arch.Msp430Arch'>
```

ppci.api.**get_current_arch**()
Try to get the architecture for the current platform

ppci.api.**is_platform_supported**()
Determine if this platform is supported

ppci.api.**ir_to_object**(*ir_modules*, *march*, *reporter=None*, *debug=False*, *opt='speed'*, *outstream=None*)
Translate IR-modules into code for the given architecture.

Parameters

- **ir_modules** – a collection of ir-modules that will be transformed into machine code.
- **march** – the architecture for which to compile.
- **reporter** – reporter to write compilation report to
- **debug** (*bool*) – include debugging information
- **opt** (*str*) – optimization goal. Can be 'speed', 'size' or 'co2'.
- **outstream** – instruction stream to write instructions to

Returns An object file

Return type *ObjectFile*

ppci.api.**ir_to_python**(*ir_modules*, *f*, *reporter=None*)
Convert ir-code to python code

ppci.api.**ir_to_assembly**(*ir_modules*, *march*, *add_binary=False*)
Translate the given ir-code into assembly code.

ppci.api.**bf_to_ir**(*source*, *target*)
Compile brainfuck source into ir code

ppci.api.**ws_to_ir**(*source*)
Compile whitespace source

3.2 Program classes

The Program classes provide a high level interface for working with PPCI. Each class represents one language / code representation. They have a common API to get reporting, compile into other program representations, and export to e.g. textual or binary representations.

3.2.1 Base program classes

class ppci.programs.**Program**(**items*, *previous=None*, *debugdb=None*)
Abstract base class to represent a computer program. Subclasses represent languages (i.e. code representations), e.g. Python, IR, or X86. Program objects can be compiled into one another using the `to_xx()` methods.

Each instance can have multiple "items", which can represent files or modules, and which can in some cases be bound/linked into a single object. Some Program classes also provide optimizations. Many languages can be represented in textual or binary form, and can be imported/exported as such.

Each subclass needs to implement:

- A docstring with a brief description of the language.
- Method *_check_items(items)* to test input at initialization.
- Method `_copy()`.
- Method `_get_report(html)`.

Each subclasses should implement as applicable:

- Method *optimize()*.
- Export methods like `as_text()`.
- Import classmethods like `from_text()`.

chain
A tuple with the names of the languages that the current object originated from.

copy()
Make a (deep) copy of the program.

get_report(*html=False*)
Get a textual representation of the program for introspection and debugging purposes. If `html` the report may be html-formatted.

items
The list of items, representing components such as files or modules.

previous(*which=1*)
Get a previous Program instance, or None.

Parameters **which** –

- int: Go this many steps back in the compile chain (default 1).
- str: Get the program in the compile chain that represents the given language.
- Program instance/class: Get the program in the compile chain that represents the given Program class.

source
The toplevel Program instance that is the source of the compile chain.

to(*language*, ***options*)
Compile this program into another representation. The tree is traversed to find the lowest cost (i.e. shortest) chain of compilers to the target language.

Experimental; use with care.

class ppci.programs.**SourceCodeProgram**(**items*, *previous=None*, *debugdb=None*)
Base class for source code.

(i.e. intended to be read and written by humans).

get_tokens()
Get the program in the form of a series of (standardized) tokens, for the purpose of syntax highlighting.

class ppci.programs.**IntermediateProgram**(**items*, *previous=None*, *debugdb=None*)
Base class for intermediate code representations.

These are programs that are not human readable nor machine executable.

class ppci.programs.**MachineProgram**(**items*, *previous=None*, *debugdb=None*)
Base class for executable machine code.

run_in_process()
If the architecture of the code matches the current machine, execute the code in this Python process.

3.2.2 Source code programs

class ppci.programs.**C3Program**(**items*, *previous=None*, *debugdb=None*)
C3 is a novel programming language ispired by C, but avoiding some of its contraptions.

The items in a C3Program are strings.

to_ir(*includes=None*, *march=None*, *reporter=None*)
Compile C3 to PPCI IR for the given architecture.

class ppci.programs.**PythonProgram**(**items*, *previous=None*, *debugdb=None*)
Python is a dynamic programming language, which is popular due to its great balance between simpliciy and expressiveness, and a thriving community.

The items in a PythonProgram are strings.

run(*namespace=None*)
Run (i.e. exec()) the code in the current interpreter.

to_ir()
Compile Python to PPCI IR.

Status: very preliminary.

to_wasm()
Compile Python to WASM.

Status: can compile a subset of Python, and assumes all floats.

3.2.3 Intermediate programs

class ppci.programs.**IrProgram**(**items*, *previous=None*, *debugdb=None*)
PPCI IR code is the intermediate representation used in PPCI.

optimize(*level=2*)
Optimize the ir program

to_arm(***options*)
Compile to ARM machine code.

Status: ...

to_python(***options*)
Compile PPCI IR to Python. Not very efficient or pretty code, but it can be useful to test the IR code without compiling to machine code.

Status: complete: can compile the full IR spec.

to_wasm(***options*)
Compile PPCI IR to WASM.

Do this by taking each ir module into a wasm module.

to_x86(***options*)
Compile to X86 machine code.

Status: ...

class ppci.programs.**WasmProgram**(**items*, *previous=None*, *debugdb=None*)
WASM (a.k.a. Web Assembly) is an open standard to represent code in a compact, low level format that can be easily converterted to machine code, and run fast as well as safe.

Items in a WasmProgram are WasmModule objects.

as_bytes()
Convert to WASM binary representation.

as_hex()
Turn into a hex representation (using either the byte representation or the text representation). Raises NotImplementedError if this program does not have a binary nor textual representation.

to_ir(***options*)
Compile WASM to IR.

Status: very basic.

3.2.4 Machine code programs

class ppci.programs.**ArmProgram**(**items*, *previous=None*, *debugdb=None*)
Machine code for most mobile devices and e.g. the Raspberry Pi.

as_object()
Export as binary code object (bytes)

class ppci.programs.**X86Program**(**items*, *previous=None*, *debugdb=None*)
Machine code for most common desktops and laptops.

as_elf(*filename*)
Export as elf file.

as_exe(*filename*)
Export as a system executable file.

as_object()
Export as binary code object (bytes)

3.3 Command line tools

This section describes the usage the commandline tools installed with ppci.

Take for example the stm32f4 blinky project. To build this project, run ppci-build.py in the project folder:

```
$ cd examples/blinky
$ ppci-build
```

This command is used to construct *build files*.

Or specify the buildfile a the command line:

```
$ ppci-build -f examples/blinky/build.xml
```

Instead of relying on a build system, the *c3* compiler can also be activated stand alone.

```
$ ppci-c3c --machine arm examples/snake/game.c3
```

3.3.1 ppci-c3c

C3 compiler.

Use this compiler to produce object files from c3 sources and c3 includes. C3 includes have the same format as c3 source files, but do not result in any code.

```
usage: ppci-c3c [-h] [--log log-level] [--report report-file]
                [--html-report html-report-file]
                [--text-report text-report-file] [--verbose] [--version]
                [--pudb]
                [--machine {arm,avr,example,m68k,mcs6500,microblaze,mips,msp430,
↪or1k,riscv,stm8,x86_64,xtensa}]
                [--mtune option] [--output output-file] [-g] [-S] [--ir]
                [--wasm] [--pycode] [-O {0,1,2,s}] [--instrument-functions]
                [-i include]
                source [source ...]
```

source
source file

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

--machine, -m
target architecture

--mtune <option>
architecture option

--output <output-file>, **-o** <output-file>
output file

-g
create debug information

-S
Do not assemble, but output assembly language

--ir
Output ppci ir-code, do not generate code

--wasm
Output WASM (WebAssembly)

--pycode
Output python code

-O {0,1,2,s}
optimize code

--instrument-functions
Instrument given functions

-i <include>, **--include** <include>
include file

3.3.2 ppci-build

Build utility.

Use this to execute build files.

```
usage: ppci-build [-h] [--log log-level] [--report report-file]
                  [--html-report html-report-file]
                  [--text-report text-report-file] [--verbose] [--version]
                  [--pudb] [-f build-file]
                  [target [target ...]]
```

target

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

-f <build-file>, **--buildfile** <build-file>
use buildfile, otherwise build.xml is the default

3.3.3 ppci-archive

Archive manager.

Create or update an archive. Or extract object files from the archive.

```
usage: ppci-archive [-h] [--log log-level] [--report report-file]
                    [--html-report html-report-file]
                    [--text-report text-report-file] [--verbose] [--version]
                    [--pudb]
                    {create,display} ...
```

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

ppci-archive create

```
usage: ppci-archive create [-h] archive [obj [obj ...]]
```

archive
Archive filename.

obj
the object to link

-h, --help
show this help message and exit

ppci-archive display

```
usage: ppci-archive display [-h] archive
```

archive
Archive filename.

-h, --help
show this help message and exit

3.3.4 ppci-asm

Assembler utility.

```
usage: ppci-asm [-h] [--log log-level] [--report report-file]
                [--html-report html-report-file]
                [--text-report text-report-file] [--verbose] [--version]
                [--pudb]
                [--machine {arm,avr,example,m68k,mcs6500,microblaze,mips,msp430,
↪or1k,riscv,stm8,x86_64,xtensa}]
                [--mtune option] [--output output-file] [-g]
                sourcefile
```

sourcefile
the source file to assemble

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

--machine, -m
target architecture

--mtune <option>
architecture option

--output <output-file>, **-o** <output-file>
output file

-g, --debug
create debug information

3.3.5 ppci-ld

Linker.

Use the linker to combine several object files and a memory layout to produce another resulting object file with images.

```
usage: ppci-ld [-h] [--log log-level] [--report report-file]
               [--html-report html-report-file]
               [--text-report text-report-file] [--verbose] [--version]
               [--pudb] [--output output-file] [--library library-filename]
               [--layout layout-file] [-g] [--relocatable] [--entry ENTRY]
               obj [obj ...]
```

obj
the object to link

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

--output <output-file>, **-o** <output-file>
output file

--library <library-filename>
Add library to use when searching for symbols.

--layout <layout-file>, **-L** <layout-file>
memory layout

-g
retain debug information

--relocatable, -r
Generate relocatable output

--entry <entry>, **-e** <entry>
Use entry as the starting symbol of execution of the program.

3.3.6 ppci-objcopy

Objcopy utility to manipulate object files.

```
usage: ppci-objcopy [-h] [--log log-level] [--report report-file]
                    [--html-report html-report-file]
                    [--text-report text-report-file] [--verbose] [--version]
                    [--pudb] [--segment SEGMENT]
                    [--output-format OUTPUT_FORMAT]
                    input output
```

input
input file

output
output file

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

--segment , **-S**
segment to copy

--output-format <output_format>, **-O** <output_format>
output file format

3.3.7 ppci-objdump

Objdump utility to display the contents of object files.

```
usage: ppci-objdump [-h] [--log log-level] [--report report-file]
                    [--html-report html-report-file]
                    [--text-report text-report-file] [--verbose] [--version]
                    [--pudb] [-d]
                    obj
```

obj
object file

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

-d, --disassemble
Disassemble contents

3.3.8 ppci-opt

Optimizer

```
usage: ppci-opt [-h] [--log log-level] [--report report-file]
                [--html-report html-report-file]
                [--text-report text-report-file] [--verbose] [--version]
                [--pudb] [-O O]
                input output
```

input
input file

output
output file

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

-O <o>
Optimization level

3.3.9 ppci-cc

C compiler.

Use this compiler to compile C source code to machine code for different computer architectures.

```
usage: ppci-cc [-h] [--log log-level] [--report report-file]
               [--html-report html-report-file]
               [--text-report text-report-file] [--verbose] [--version]
               [--pudb]
               [--machine {arm,avr,example,m68k,mcs6500,microblaze,mips,msp430,
↪or1k,riscv,stm8,x86_64,xtensa}]
               [--mtune option] [--output output-file] [-g] [-S] [--ir]
               [--wasm] [--pycode] [-O {0,1,2,s}] [--instrument-functions]
               [-I dir] [-D macro] [-U macro] [--include file] [--trigraphs]
               [--std {c89,c99}] [--super-verbose] [--freestanding] [-E] [-M]
               [--ast] [-c]
               source [source ...]
```

source
source file

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

--machine, -m
target architecture

--mtune <option>
architecture option

--output <output-file>, **-o** <output-file>
output file

-g
create debug information

-S
Do not assemble, but output assembly language

--ir
Output ppci ir-code, do not generate code

--wasm
Output WASM (WebAssembly)

--pycode
Output python code

-O {0,1,2,s}
optimize code

--instrument-functions
Instrument given functions

-I <dir>
Add directory to the include path

-D <macro>, **--define** <macro>
Define a macro

-U <macro>, **--undefine** <macro>
Undefine a macro

--include <file>
Include a file before all other sources

--trigraphs
Enable trigraph processing

--std {c89,c99}
The C version you want to use

--super-verbose
Add extra verbose output during C compilation

--freestanding
Compile in free standing mode.

-E
Stop after preprocessing

-M
Instead of preprocessing, emit a makefile rule with dependencies

--ast
Stop parsing and output the C abstract syntax tree (ast)

-c
Compile, but do not link

3.3.10 ppci-pascal

Pascal compiler.

Compile pascal programs.

```
usage: ppci-pascal [-h] [--log log-level] [--report report-file]
                   [--html-report html-report-file]
                   [--text-report text-report-file] [--verbose] [--version]
                   [--pudb]
                   [--machine {arm,avr,example,m68k,mcs6500,microblaze,mips,msp430,
↪or1k,riscv,stm8,x86_64,xtensa}]
                   [--mtune option] [--output output-file] [-g] [-S] [--ir]
                   [--wasm] [--pycode] [-O {0,1,2,s}] [--instrument-functions]
                   source [source ...]
```

source
source file

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

--machine, -m
target architecture

--mtune <option>
architecture option

--output <output-file>, **-o** <output-file>
output file

-g
create debug information

-S
Do not assemble, but output assembly language

--ir
Output ppci ir-code, do not generate code

--wasm
Output WASM (WebAssembly)

--pycode
Output python code

-O {0,1,2,s}
optimize code

--instrument-functions
Instrument given functions

3.3.11 ppci-pycompile

Compile python code statically

```
usage: ppci-pycompile [-h] [--log log-level] [--report report-file]
                      [--html-report html-report-file]
                      [--text-report text-report-file] [--verbose] [--version]
                      [--pudb]
                      [--machine {arm,avr,example,m68k,mcs6500,microblaze,mips,
↪msp430,or1k,riscv,stm8,x86_64,xtensa}]
```

(continues on next page)

(continued from previous page)

```
                    [--mtune option] [--output output-file] [-g] [-S] [--ir]
                    [--wasm] [--pycode] [-O {0,1,2,s}]
                    [--instrument-functions]
                    source [source ...]
```

source
source file

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

--machine, -m
target architecture

--mtune <option>
architecture option

--output <output-file>, **-o** <output-file>
output file

-g
create debug information

-S
Do not assemble, but output assembly language

--ir
Output ppci ir-code, do not generate code

--wasm
Output WASM (WebAssembly)

--pycode
Output python code

-O {0,1,2,s}
optimize code

--instrument-functions
Instrument given functions

3.3.12 ppci-readelf

Clone of the famous *readelf* utility

```
usage: ppci-readelf [-h] [--log log-level] [--report report-file]
                    [--html-report html-report-file]
                    [--text-report text-report-file] [--verbose] [--version]
                    [--pudb] [-a] [--file-header] [-l] [-S] [-s] [-e]
                    [-x HEX_DUMP] [--debug-dump {rawline,}]
                    elf
```

elf
ELF file

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

-a, --all
Equivalent to: -h -l -S -s -r -d -V -A -I

--file-header
Display the ELF file header

-l, --program-headers
Display the program headers

-S, --section-headers
Display the section headers

-s, --syms
Display the symbol table

-e, --headers
Equivalent to: –file-header -l -S

-x <hex_dump>, **--hex-dump** <hex_dump>
Dump contents of section as bytes

--debug-dump {rawline,}
Display contents of dwarf sections

3.3.13 ppci-wasmcompile

Static web assembly compiler.

This command line tool takes web assembly to native code.

```
usage: ppci-wasmcompile [-h] [--log log-level] [--report report-file]
                        [--html-report html-report-file]
                        [--text-report text-report-file] [--verbose]
                        [--version] [--pudb]
                        [--machine {arm,avr,example,m68k,mcs6500,microblaze,mips,
↪msp430,or1k,riscv,stm8,x86_64,xtensa}]
                        [--mtune option] [--output output-file] [-g] [-S]
                        [--ir] [--wasm] [--pycode] [-O {0,1,2,s}]
                        [--instrument-functions]
                        wasm file
```

wasm file
: wasm file to compile

-h, --help
: show this help message and exit

--log <log-level>
: Log level (info,debug,warn)

--report
: Specify a file to write the compile report to

--html-report
: Write html report file

--text-report
: Write a report into a text file

--verbose, -v
: Increase verbosity of the output

--version, -V
: Display version and exit

--pudb
: Drop into post mortem pudb session on crash

--machine, -m
: target architecture

--mtune <option>
: architecture option

--output <output-file>, **-o** <output-file>
: output file

-g
: create debug information

-S
: Do not assemble, but output assembly language

--ir
: Output ppci ir-code, do not generate code

--wasm
: Output WASM (WebAssembly)

--pycode
: Output python code

-O {0,1,2,s}
: optimize code

--instrument-functions
: Instrument given functions

3.3.14 ppci-yacc

Parser generator utility.

This script can generate a python script from a grammar description.

Invoke the script on a grammar specification file:

```
$ ppci-yacc test.x -o test_parser.py
```

And use the generated parser by deriving a user class:

```
import test_parser
class MyParser(test_parser.Parser):
    pass
p = MyParser()
p.parse()
```

Alternatively you can load the parser on the fly:

```
import yacc
parser_mod = yacc.load_as_module('mygrammar.x')
class MyParser(parser_mod.Parser):
    pass
p = MyParser()
p.parse()
```

```
usage: ppci-yacc [-h] [--log log-level] [--report report-file]
                 [--html-report html-report-file]
                 [--text-report text-report-file] [--verbose] [--version]
                 [--pudb] -o OUTPUT
                 source
```

source
the parser specification

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

-o <output>, **--output** <output>

3.3.15 ppci-wasm2wat

Convert binary wasm to wasm text (WAT) format.

```
usage: ppci-wasm2wat [-h] [--log log-level] [--report report-file]
                     [--html-report html-report-file]
                     [--text-report text-report-file] [--verbose] [--version]
                     [--pudb] [-o wat file]
                     wasm file
```

wasm file
wasm file to read

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

-o <wat file>, **--output** <wat file>
File to write the WAT file to, default is stdout

3.3.16 ppci-wat2wasm

Convert binary wasm to wasm text (WAT) format.

```
usage: ppci-wat2wasm [-h] [--log log-level] [--report report-file]
                     [--html-report html-report-file]
                     [--text-report text-report-file] [--verbose] [--version]
                     [--pudb] [-o wasm file]
                     wat file
```

wat file
wasm text file to read

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

-o <wasm file>, **--output** <wasm file>
File to write the binary wasm file to, default is stdout

3.3.17 ppci-wabt

Wasm binary toolkit (WABT)

```
usage: ppci-wabt [-h] [--log log-level] [--report report-file]
                 [--html-report html-report-file]
                 [--text-report text-report-file] [--verbose] [--version]
                 [--pudb]
                 {wat2wasm,wasm2wat,show_interface,run} ...
```

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

ppci-wabt run

```
usage: ppci-wabt run [-h] [--func-arg arg] [--target target]
                     [--function function_name]
                     wasm-file [arg [arg ...]]
```

wasm-file
wasm file to run

arg
Command line argument to wasm accessible via WASI.

-h, --help
show this help message and exit

--func-arg <arg>
Argument to wasm function

--target {native,python}
Which target to generate code for

--function <function_name>, **-f** <function_name>
Function to run. Overrides WASI _start function.

ppci-wabt show_interface

```
usage: ppci-wabt show_interface [-h] wasm-file
```

wasm-file
wasm file to read

-h, --help
show this help message and exit

ppci-wabt wasm2wat

```
usage: ppci-wabt wasm2wat [-h] [-o wat file] wasm file
```

wasm file
wasm file to read

-h, --help
show this help message and exit

-o <wat file>, **--output** <wat file>
File to write the WAT file to, default is stdout

ppci-wabt wat2wasm

```
usage: ppci-wabt wat2wasm [-h] [-o wasm file] wat file
```

wat file
wasm text file to read

-h, --help
show this help message and exit

-o <wasm file>, **--output** <wasm file>
File to write the binary wasm file to, default is stdout

3.3.18 ppci-ocaml

OCaml utility.

Multiple usage possible, for example:

```
usage: ppci-ocaml [-h] [--log log-level] [--report report-file]
                  [--html-report html-report-file]
                  [--text-report text-report-file] [--verbose] [--version]
                  [--pudb]
                  {disassemble,opt} ...
```

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

ppci-ocaml disassemble

```
usage: ppci-ocaml disassemble [-h] bytecode-file
```

bytecode-file
OCaml bytecode file to disassemble

-h, --help
show this help message and exit

ppci-ocaml opt

```
usage: ppci-ocaml opt [-h]
                      [--machine {arm,avr,example,m68k,mcs6500,microblaze,mips,
↪msp430,or1k,riscv,stm8,x86_64,xtensa}]
                      [--mtune option] [--output output-file] [-g] [-S] [--ir]
                      [--wasm] [--pycode] [-O {0,1,2,s}]
                      [--instrument-functions]
                      bytecode-file
```

bytecode-file
OCaml bytecode file to disassemble

-h, --help
show this help message and exit

--machine, -m
target architecture

--mtune <option>
architecture option

--output <output-file>, **-o** <output-file>
output file

-g
create debug information

-S
Do not assemble, but output assembly language

--ir
Output ppci ir-code, do not generate code

--wasm
Output WASM (WebAssembly)

--pycode
Output python code

-O {0,1,2,s}
optimize code

--instrument-functions
Instrument given functions

3.3.19 ppci-java

Java handling utility.

```
usage: ppci-java [-h] [--log log-level] [--report report-file]
                 [--html-report html-report-file]
                 [--text-report text-report-file] [--verbose] [--version]
                 [--pudb]
                 {compile,javap,jar} ...
```

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

ppci-java compile

```
usage: ppci-java compile [-h] [--output output-file] [-g] [-S] [--ir] [--wasm]
                         [--pycode] [-O {0,1,2,s}] [--instrument-functions]
                         [--machine {arm,avr,example,m68k,mcs6500,microblaze,mips,
↪msp430,or1k,riscv,stm8,x86_64,xtensa}]
                         [--mtune option]
                         java class file
```

java class file
class file to compile

-h, --help
show this help message and exit

--output <output-file>, **-o** <output-file>
output file

-g
create debug information

-S
Do not assemble, but output assembly language

--ir
Output ppci ir-code, do not generate code

--wasm
Output WASM (WebAssembly)

--pycode
Output python code

-O {0,1,2,s}
optimize code

--instrument-functions
Instrument given functions

--machine, -m
target architecture

--mtune <option>
architecture option

ppci-java jar

```
usage: ppci-java jar [-h] java jar file
```

java jar file
jar file to inspect

-h, --help
show this help message and exit

ppci-java javap

```
usage: ppci-java javap [-h] java class file
```

java class file
class file to inspect

-h, --help
show this help message and exit

3.3.20 ppci-hexutil

hexfile manipulation tool by Windel Bouwman

```
usage: ppci-hexutil [-h] {info,new,merge} ...
```

-h, --help
show this help message and exit

ppci-hexutil info

```
usage: ppci-hexutil info [-h] hexfile
```

hexfile

-h, --help
show this help message and exit

ppci-hexutil merge

```
usage: ppci-hexutil merge [-h] hexfile1 hexfile2 rhexfile
```

hexfile1
hexfile 1

hexfile2
hexfile 2

rhexfile
resulting hexfile

-h, --help
show this help message and exit

ppci-hexutil new

```
usage: ppci-hexutil new [-h] hexfile address datafile
```

hexfile

address
hex address of the data

datafile
binary file to add

-h, --help
show this help message and exit

3.3.21 ppci-hexdump

Display file contents in hexadecimal

```
usage: ppci-hexdump [-h] [--log log-level] [--report report-file]
                    [--html-report html-report-file]
                    [--text-report text-report-file] [--verbose] [--version]
                    [--pudb] [--width WIDTH]
                    file
```

file
File to dump contents of

-h, --help
show this help message and exit

--log <log-level>
Log level (info,debug,warn)

--report
Specify a file to write the compile report to

--html-report
Write html report file

--text-report
Write a report into a text file

--verbose, -v
Increase verbosity of the output

--version, -V
Display version and exit

--pudb
Drop into post mortem pudb session on crash

--width <width>
Width of the hexdump.

3.4 Languages

This section describes the various modules which deal with programming languages.

3.4.1 Basic

This submodule supports the basic programming language.

Warning: This module is a work in progress.

Basic languages module

3.4.2 Brainfuck

The compiler has a front-end for the brainfuck language. You can use *bf_to_ir()* to transform brainfuck code into IR-code:

```
>>> from ppci.lang.bf import bf_to_ir
>>> import io
>>> ir_module = bf_to_ir(io.StringIO('>>ignore.'), 'arm')
>>> ir_module.display()
module main;

external procedure bsp_putc(u8);

global variable data (30000 bytes aligned at 4)

global procedure main() {
  main_block0: {
    blob<4:4> ptr_alloc = alloc 4 bytes aligned at 4;
    ptr ptr_addr = &ptr_alloc;
    i32 num = 1;
    i8 val_inc = cast num;
    ptr ptr_incr = cast num;
    i32 num_0 = 0;
    ptr zero_ptr = cast num_0;
    i8 zero_ptr_1 = cast num_0;
    ptr num_2 = 30000;
    store zero_ptr, ptr_addr;
    jmp main_block2;
  }

  main_block1: {
    exit;
  }
```

(continues on next page)

(continued from previous page)

```
  main_block2: {
    ptr tmp_load = load ptr_addr;
    ptr tmp = data + tmp_load;
    store num_0, tmp;
    ptr tmp_3 = tmp_load + ptr_incr;
    store tmp_3, ptr_addr;
    cjmp tmp_3 == num_2 ? main_block1 : main_block2;
  }

}
```

Reference

This is the brain-fuck language front-end.

ppci.lang.bf.**bf_to_ir**(*source*, *target*)
Compile brainfuck source into ir code

class ppci.lang.bf.**BrainFuckGenerator**(*arch*)
Brainfuck is a language that is so simple, the entire front-end can be implemented in one pass.

generate(*src*, *module_name='main'*, *function_name='main'*)
Takes a brainfuck program and returns the IR-code module

3.4.3 C3 language

Introduction

As an example of designing and implementing a custom language within the PPCI framework, the C3 language was created. As pointed out in c2lang, the C language is widely used, but has some strange contraptions. These include the following:

- The include system. This results in lots of code duplication and file creation. Why would you need filenames in source code?
- The comma statement: x = a(), 2; assigns 2 to x, after calling function a.
- C is difficult to parse with a simple parser. The parser has to know what a symbol is when it is parsed. This is also referred to as the lexer hack.

In part for these reasons (and of course, for fun), C3 was created.

The hello world example in C3 is:

```
module hello;
import io;

function void main()
{
    io.println("Hello world");
}
```

Language reference

Modules

Modules in C3 live in file, and can be defined in multiple files. Modules can import each other by using the `import` statement.

For example:

pkg1.c3:

```
module pkg1;
import pkg2;
```

pkg2.c3:

```
module pkg2;
import pkg1;
```

Functions

Function can be defined by using the `function` keyword, followed by a type and the function name.

```
module example;

function void compute()
{
}

function void main()
{
    main();
}
```

Variables

Variables require the `var` keyword, and can be either global or function-local.

```
module example;

var int global_var;

function void compute()
{
    var int x = global_var + 13;
    global_var = 200 - x;
}
```

Types

Types can be specified when a variable is declared, and also typedef'ed using the `type` keyword.

```
module example;
var int number;
var int* ptr_num;
type int* ptr_num_t;
var ptr_num_t number2;
```

If statement

The following code example demonstrates the `if` statement. The `else` part is optional.

```
module example;

function void compute(int a)
{
    var int b = 10;
    if (a > 100)
    {
        b += a;
    }

    if (b > 50)
    {
        b += 1000;
    }
    else
    {
        b = 2;
    }
}
```

While statement

The `while` statement can be used as follows:

```
module example;

function void compute(int a)
{
    var int b = 10;
    while (b > a)
    {
        b -= 1;
    }
}
```

For statement

The `for` statement works like in C. The first item is initialized before the loop. The second is the condition for the loop. The third part is executed when one run of the loop is done.

```
module example;

function void compute(int a)
{
    var int b = 0;
    for (b = 100; b > a; b -= 1)
    {
        // Do something here!
    }
}
```

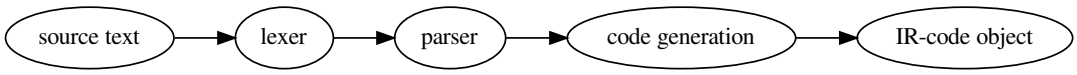
Other

C3 does not contain a preprocessor. For these kind of things it might be better to use a templating engine such as Jinja2.

Module reference

This is the c3 language front end.

For the front-end a recursive descent parser is created.



class ppci.lang.c3.**AstPrinter**
Prints an AST as text

class ppci.lang.c3.**C3Builder**(*diag*, *arch_info*)
Generates IR-code from c3 source.

Reports errors to the diagnostics system.

build(*sources*, *imps=()*)
Create IR-code from sources.

Returns A context where modules are living in and an ir-module.

Raises compiler error when something goes wrong.

do_parse(*src*, *context*)
Lexing and parsing stage (phase 1)

class ppci.lang.c3.**CodeGenerator**(*diag*)
Generates intermediate (IR) code from a package.

The entry function is 'genModule'. The main task of this part is to rewrite complex control structures, such as while and for loops into simple conditional jump statements. Also complex conditional statements are simplified. Such as 'and' and 'or' statements are rewritten in conditional jumps. And structured datatypes are rewritten.

Type checking is done in one run with code generation.

emit(*instruction*, *loc=None*)
Emits the given instruction to the builder.

error(*msg*, *loc=None*)
Emit error to diagnostic system and mark package as invalid

gen(*context*)
Generate code for a whole context

gen_assignment_stmt(*code*)
Generate code for assignment statement

gen_binop(*expr: ppci.lang.c3.astnodes.Binop*)
Generate code for binary operation

gen_bool_expr(*expr*)
Generate code for cases where a boolean value is assigned

gen_cond_code(*expr*, *bbtrue*, *bbfalse*)
Generate conditional logic. Implement sequential logical operators.

gen_dereference(*expr: ppci.lang.c3.astnodes.Deref*)
dereference pointer type, which means *(expr)

gen_expr_at(*ptr*, *expr*)
Generate code at a pointer in memory

gen_expr_code(*expr: ppci.lang.c3.astnodes.Expression*, *rvalue=False*) → ppci.ir.Value
Generate code for an expression. Return the generated ir-value

gen_external_function(*function*)
Generate external function

gen_for_stmt(*code*)
Generate for-loop code

gen_function(*function*)
Generate code for a function. This involves creating room for parameters on the stack, and generating code for the function body.

gen_function_call(*expr*)
Generate code for a function call

gen_global_ival(*ival*, *typ*)
Create memory image for initial value

gen_globals(*module*)
Generate global variables and modules

gen_identifier(*expr*)
Generate code for when an identifier was referenced

gen_if_stmt(*code*)
Generate code for if statement

gen_index_expr(*expr*)
Array indexing

gen_literal_expr(*expr*)
Generate code for literal

gen_local_var_init(*var*)
Initialize a local variable

gen_member_expr(*expr*)
Generate code for member expression such as struc.mem = 2 This could also be a module deref!

gen_module(*mod: ppci.lang.c3.astnodes.Module*)
Generate code for a single module

gen_return_stmt(*code*)
Generate code for return statement

gen_stmt(*code: ppci.lang.c3.astnodes.Statement*)
Generate code for a statement

gen_switch_stmt(*switch*)
Generate code for a switch statement

gen_type_cast(*expr*)
Generate code for type casting

gen_unop(*expr*)
Generate code for unary operator

gen_while(*code*)
Generate code for while statement

get_debug_type(*typ*)
Get or create debug type info in the debug information

get_ir_function(*function*)
Get the proper IR function for the given function.

A new function will be created if required.

get_ir_type(*cty*)
Given a certain type, get the corresponding ir-type

is_module_ref(*expr*)
Determine whether a module is referenced

new_block()
Create a new basic block into the current function

class ppci.lang.c3.**Context**(*arch_info*)
A context is the space where all modules live in.

It is actually the container of modules and the top level scope.

equal_types(*a*, *b*, *byname=False*)
Compare types a and b for structural equavalence.

if byname is True stop on defined types.

eval_const(*expr*)
Evaluates a constant expression.

get_common_type(*a*, *b*, *loc*)
Determine the greatest common type.

This is used for coercing binary operators.

For example:

- int + float -> float
- byte + int -> int
- byte + byte -> byte
- pointer to x + int -> pointer to x

get_constant_value(*const*)
Get the constant value, calculate if required

get_module(*name*, *create=True*)
Gets or creates the module with the given name

get_type(*typ*, *reveil_defined=True*)
Get type given by str, identifier or type.

When reveil_defined is True, defined types are resolved to their backing types.

has_module(*name*)
Check if a module with the given name exists

is_simple_type(*typ*)
Determines if the given type is a simple type

link_imports()
Resolve all modules referenced by other modules

modules
Get all the modules in this context

pack_string(*txt*)
Pack a string an int as length followed by text data

resolve_symbol(*ref*)
Find out what is designated with x

size_of(*typ*)
Determine the byte size of a type

class ppci.lang.c3.**Lexer**(*diag*)
Generates a sequence of token from an input stream

tokenize(*text*)
Keeps track of the long comments

class ppci.lang.c3.**Parser**(*diag*)
Parses sourcecode into an abstract syntax tree (AST)

add_symbol(*sym*)
Add a symbol to the current scope

parse_cast_expression() → ppci.lang.c3.astnodes.Expression
Parse a cast expression.

The C-style type cast conflicts with '(' expr ')' so introduce extra keyword 'cast'.

parse_compound()
Parse a compound statement, which is bounded by '{' and '}'

parse_const_def()
Parse a constant definition

parse_const_expression()
Parse array initializers and other constant values

parse_designator()
A designator designates an object with a name.

parse_expression(*rbp=0*) → ppci.lang.c3.astnodes.Expression
Process expressions with precedence climbing.

See also:

http://eli.thegreenplace.net/2012/08/02/ parsing-expressions-by-precedence-climbing

parse_for() → ppci.lang.c3.astnodes.For
Parse a for statement

parse_function_def(*public=True*)
Parse function definition

parse_id_sequence()
Parse a sequence of id's

parse_if()
Parse if statement

parse_import()
Parse import construct

parse_module(*context*)
Parse a module definition

parse_postfix_expression() → ppci.lang.c3.astnodes.Expression
Parse postfix expression

parse_primary_expression() → ppci.lang.c3.astnodes.Expression
Literal and parenthesis expression parsing

parse_return() → ppci.lang.c3.astnodes.Return
Parse a return statement

parse_source(*tokens*, *context*)
Parse a module from tokens

parse_statement() → ppci.lang.c3.astnodes.Statement
Determine statement type based on the pending token

parse_switch() → ppci.lang.c3.astnodes.Switch
Parse switch statement

parse_top_level()
Parse toplevel declaration

parse_type_def(*public=True*)
Parse a type definition

parse_type_spec()
Parse type specification. Type specs are read from right to left.

A variable spec is given by: var [typeSpec] [modifiers] [pointer/array suffix] variable_name

For example: var int volatile * ptr; creates a pointer to a volatile integer.

parse_unary_expression()
Handle unary plus, minus and pointer magic

parse_variable_def(*public=True*)
Parse variable declaration, optionally with initialization.

parse_while() → ppci.lang.c3.astnodes.While
Parses a while statement

class ppci.lang.c3.**Visitor**(*pre=None*, *post=None*)
Visitor that can visit all nodes in the AST and run pre and post functions.

do(*node*)
Visit a single node

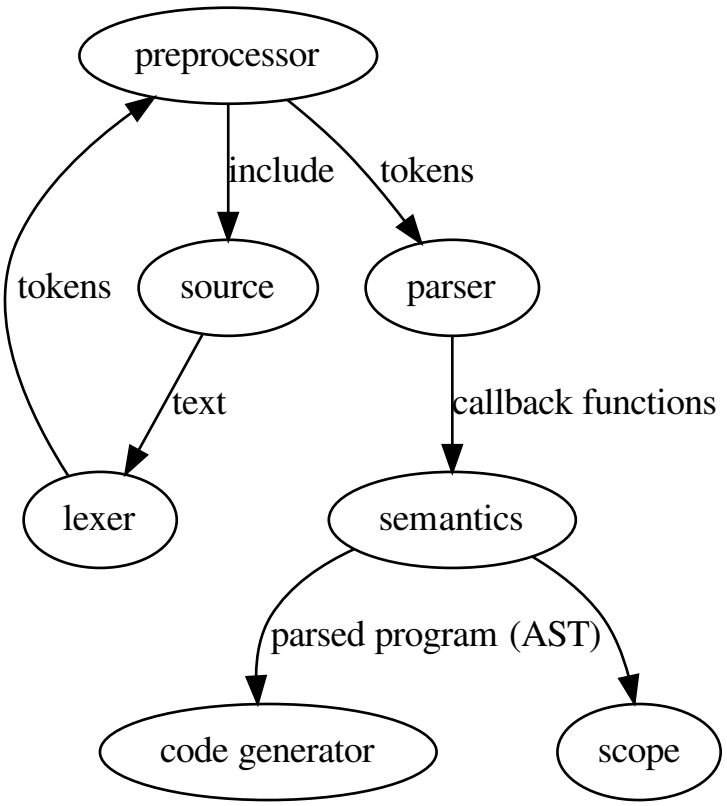
visit(*node*)
Visit a node and all its descendants

ppci.lang.c3.**c3_to_ir**(*sources*, *includes*, *march*, *reporter=None*)
Compile c3 sources to ir-code for the given architecture.

3.4.4 C compiler

This page describes the design of the C compiler. If you want to use the compiler, please see *ppci-cc* or *ppci.api.cc()*.

Overview



The C compilers task is to take C code, and produce intermediate code for the backend. The first step is pre processing, during which a sequence of tokens is transformed into another sequence of tokens. Next comes parsing and code generation. There are two options here: split the parser and code generation passes, such that there is a clean seperation, or do it all at once. Maybe the best choice is to do it all in one pass, and transform a sequence of tokens into IR-code in a single go.

Limitations

The C compiler has some limitations. This is the list of current known limitations:

- type qualifiers (const, volatile, etc) are parsed but ignored
- K&R function declarations are not supported
- wide characters are not supported
- VLA is not implemented
- function declarations in function definitions are not allowed

Pre-processor

The preprocessor is a strange thing. It must handle trigraphs, backslash newlines and macros.

The top level design of the preprocessor is the following:

- Context: Contains state that would be global otherwise.
- *CLexer*: processes a raw file into a sequence of tokens
- Preprocessor: takes the token sequence a does macro expansion, resulting in another stream of tokens.
- Output: The token stream maybe outputted to file.
- Feed to compiler: The token stream might be fed into the rest of the compiler.

C compiler

The c compiler consists of the classical stages: parsing and codegeneration. Code generation is done to ir-code.

Parsing

The C parsing is done by two classes *CParser* and *CSemantics*. CParser is a recursive descent parser. It dances a tight dance with the CSemantics class. This idea is taken from the Clang project. The CParser takes a token sequence from the preprocessor and matches the C syntax. Whenever a valid C construct is found, it calls the corresponding function on the CSemantics class. The semantics class keeps track of the current scope and records the global declarations. It also check types and lvalues of expressions. At the end of parsing and this type checking, an abstract syntax tree (AST) is build up. This AST is type checked and variables are resolved. This AST can be used for several purposes, for example static code analysis or style checking. It is also possible to generate C code again from this AST.

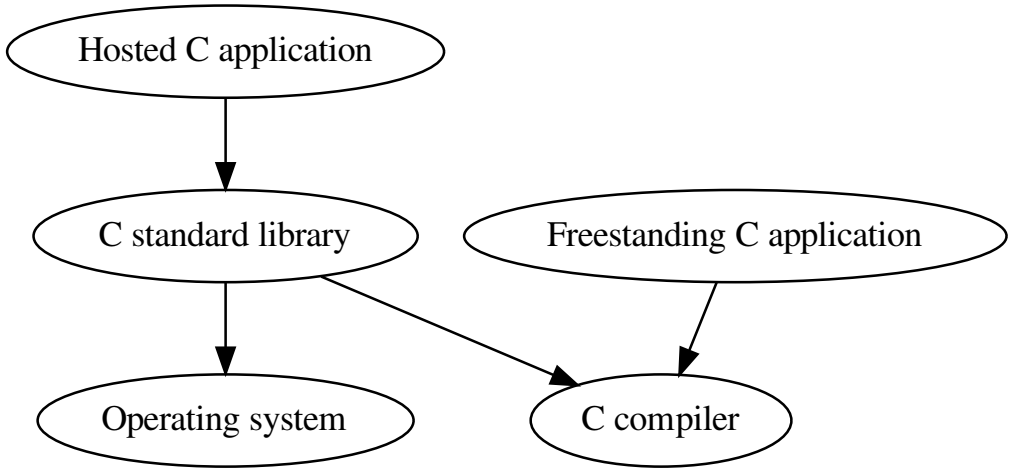
L-values

The semantics class determines for each expression node whether it is an lvalue or not. The lvalue (short for location value) indicates whether the expression has a memory address, or is a value in a register. Basically it boils down to: can we take the address of this expression with the '&' operator. Numeric literals and the results of addition are not lvalues. Variables are lvalues. The lvalue property is used during code generation to check whether the value must be loaded from memory or not.

Types

Hosted vs freestanding

A C compiler can be hosted or freestanding. The difference between those two is that a hosted C compiler also provides the standard C library. A freestanding compiler only contains a few really required header files. As a result a hosted compiler is really a combination of a C compiler and a C standard library implementation. Also, the standard library depends on the operating system which is used, where as a freestanding C compiler can be used independent of operating system. Writing an application using a hosted compiler is easier since the standard library is available.



Code generation

Code generation takes the AST as a whole and loops over all its elements and generates the corresponding IR-code snippets from it. At the end of code generation, there is an IR module which can be feed into the optimizers or code generators.

C classes

The C frontend can be used to generate an AST from C code. You can use it to parse C code and analyze its structure like this:

```
>>> from ppci.api import get_arch
>>> from ppci.lang.c import create_ast
>>> src = "int a, *b;"
>>> ast = create_ast(src, get_arch('msp430').info)
>>> ast
Compilation unit with 2 declarations
>>> from ppci.lang.c import CAstPrinter
>>> printer = CAstPrinter()
>>> printer.print(ast)
Compilation unit with 2 declarations
    Variable [storage=None typ=Basic type int name=a]
        Basic type int
    Variable [storage=None typ=Pointer-type name=b]
        Pointer-type
            Basic type int
>>> ast.declarations[0].name
'a'
```

Module reference

C front end.

ppci.lang.c.**create_ast**(*src*, *arch_info*, *filename='<snippet>'*, *coptions=None*)
Create a C ast from the given source

ppci.lang.c.**preprocess**(*f*, *output_file*, *coptions=None*)
Pre-process a file into the other file.

ppci.lang.c.**c_to_ir**(*source: io.TextIOBase*, *march*, *coptions=None*, *reporter=None*)
C to ir translation.

Parameters

- **source** (*file-like object*) – The C source to compile.
- **march** (*str*) – The targetted architecture.
- **coptions** – C specific compilation options.

Returns An *ppci.ir.Module*.

ppci.lang.c.**print_ast**(*ast*, *file=None*)
Display an abstract syntax tree.

ppci.lang.c.**parse_text**(*text*, *arch='x86_64'*)
Parse given C sourcecode into an AST

ppci.lang.c.**render_ast**(*ast*)
Render a C program as text

For example:

```
>>> from ppci.lang.c import parse_text, print_ast, render_ast
>>> ast = parse_text('int a;')
>>> print_ast(ast)
Compilation unit with 1 declarations
    Variable [storage=None typ=Basic type int name=a]
        Basic type int
>>> render_ast(ast)
int a;
```

ppci.lang.c.**parse_type**(*text*, *context*, *filename='foo.c'*)
Parse given C-type AST.

For example:

```
>>> from ppci.api import get_arch
>>> from ppci.lang.c import parse_type, CContext, COptions
>>> msp430_arch = get_arch('msp430')
>>> coptions = COptions()
>>> context = CContext(coptions, msp430_arch.info)
>>> ast = parse_type('int[2]', context)
>>> context.eval_expr(ast.size)
2
>>> context.sizeof(ast)
4
```

class ppci.lang.c.**CBuilder**(*arch_info*, *coptions*)
C builder that converts C code into ir-code

class ppci.lang.c.**CContext**(*coptions*, *arch_info*)
A context as a substitute for global data

alignment(*typ: ppci.lang.c.nodes.types.CType*)
Given a type, determine its alignment in bytes

static error(*message*, *location*, *hints=None*)
Trigger an error at the given location

eval_expr(*expr*)
Evaluate an expression right now! (=at compile time)

get_field(*typ*, *field_name*)
Get the given field.

get_field_offsets(*typ*)
Get a dictionary with offset of fields

has_field(*typ*, *field_name*)
Check if the given type has the given field.

layout_struct(*typ*)
Layout the fields in the struct.

Things to take in account: - alignment - bit packing - anonynous types

limit_max(*typ: ppci.lang.c.nodes.types.CType*) → int
Retrieve the maximum value for the given integer type.

offsetof(*typ*, *field*)
Returns the offset of a field in a struct/union in bytes

pack(*typ*, *value*)
Pack a type into proper memory format

sizeof(*typ: ppci.lang.c.nodes.types.CType*)
Given a type, determine its size in whole bytes

warning(*message*, *location*, *hints=None*)
Trigger a warning at the given location

class ppci.lang.c.**CLexer**(*coptions*)
Lexer used for the preprocessor

lex(*src*, *source_file*)
Read a source and generate a series of tokens

lex_c()
Root parsing function

lex_char()
Scan for a complete character constant

lex_float()
Lex floating point number from decimal dot onwards.

lex_number()
Lex a single numeric literal.

lex_string()
Scan for a complete string

lex_text(*txt*)
Create tokens from the given text

tokenize(*characters*)
Generate tokens from characters

class ppci.lang.c.**COptions**
A collection of settings regarding the C language

add_include_path(*path*)
Add a path to the list of include paths

add_include_paths(*paths*)
Add all the given include paths

classmethod from_args(*args*)
Create a new options object from parsed arguments.

process_args(*args*)
Given a set of parsed arguments, apply those

class ppci.lang.c.**CPreProcessor**(*coptions*)
A pre-processor for C source code

concat(*lhs*, *rhs*)
Concatenate two tokens

concatenate(*tokens*)
Handle the '##' token concatenation operator

consume(*typ=None*, *expand=True*)
Consume a token of a certain type

copy_tokens(*tokens*, *first_space*)
Copy a series of tokens.

define(*macro*)
Register a macro

define_object_macro(*name*, *text*, *protected=False*)
Define an object like macro.

define_special_macro(*name*, *handler*)
Define a spcial macro which has a callback function.

do_if(*condition*, *location*)
Handle #if/#ifdef/#ifndef.

eat_line()
Eat up all tokens until the end of the line.

This does not expand macros.

error(*msg*, *hints=None*, *loc=None*)
We hit an error condition.

eval_expr()
Evaluate an expression

expand(*macro_token*)
Expand a single token into possibly more tokens.

expand_macro(*macro*, *macro_token*)
Expand a single macro.

expand_token_sequence(*tokens*)
Macro expand a sequence of tokens.

gatherargs(*macro*)
Collect expanded arguments for macro

get_define(*name: str*)
Retrieve the given define!

handle_define_directive(*directive_token*)
Process *#define* directive.

handle_directive(*loc*)
Handle a single preprocessing directive

handle_elif_directive(*directive_token*)
Process *#elif* directive.

handle_else_directive(*directive_token*)
Process the *#else* directive.

handle_endif_directive(*directive_token*)
Process the *#endif* directive.

handle_error_directive(*directive_token*)
Process *#error* directive.

handle_if_directive(*directive_token*)
Process an *#if* directive.

handle_ifdef_directive(*directive_token*)
Handle an *#ifdef* directive.

handle_ifndef_directive(*directive_token*)
Handle an *#ifndef* directive.

handle_include_directive(*directive_token*)
Process the *#include* directive.

handle_include_next_directive(*directive_token*)
Process the *#include_next* directive.

handle_line_directive(*directive_token*)
Process *#line* directive.

handle_pragma_directive(*directive_token*)
Process *#pragma* directive.

handle_undef_directive(*directive_token*)
Process *#undef* directive.

handle_warning_directive(*directive_token*)
Process *#warning* directive.

in_hideset(*name*)
Test if the given macro is contained in the current hideset.

include(*filename*, *loc*, *use_current_dir=False*, *include_next=False*)
Turn the given filename into a series of tokens.

is_defined(*name: str*) → bool
Check if the given define is defined.

locate_include(*filename*, *loc*, *use_current_dir: bool*, *include_next*)
Determine which file to use given the include filename.

Parameters

- **loc** (–) – the location where this include is included.
- **use_current_dir** (–) – If true, look in the directory of the current file.

static make_token(*from_token*, *typ*, *value*)
Create a new token from another token.

next_token(*expand=True*)
Get next token

normalize_space(*args*)
Normalize spaces in macro expansions.

If we have space, it will be a single space.

parse_arguments()
Parse arguments for a function like macro.

- Keep track of parenthesis level.

parse_expression(*priority=0*)
Parse an expression in an #if

Idea taken from: https://github.com/shevek/jcpp/blob/master/ src/main/java/org/anarres/cpp/Preprocessor.java

parse_included_filename()
Parse filename after #include/#include_next

predefine_builtin_macros()
Define predefined macros

process_file(*f*, *filename=None*)
Process the given open file into tokens.

process_tokens()
Process a sequence of tokens into an expanded token sequence.

This function returns an tokens that must be looped over.

push_expansion(*expansion*)
Push a macro expansion on the stack.

skip_excluded_block()
Skip the block excluded by if/ifdef.

Skip tokens until we hit #endif or alike.

special_macro_counter(*macro_token*)
Implement __COUNTER__ macro

special_macro_date(*macro_token*)
Invoked when the __DATE__ macro is expanded

special_macro_file(*macro_token*)
Invoked when the __FILE__ macro is expanded

special_macro_include_level(*macro_token*)
Implement __INCLUDE_LEVEL__ macro

special_macro_line(*macro_token*)
Invoked when the __LINE__ macro is expanded

special_macro_time(*macro_token*)
Implement __TIME__ macro

stringify(*hash_token*, *snippet*, *loc*)
Handle the '#' stringify operator.

Take care of: - single space between the tokens being stringified - no spaces before first and after last token - escape double quotes of strings and backslash inside strings.

substitute_arguments(*macro*, *args*)
Return macro contents with substituted arguments.

Pay special care to # and ## operators, When an argument is used in # or ##, it is not macro expanded.

token
Peek one token ahead without taking it.

tokens_to_string(*tokens*)
Create a text from the given tokens

undefine(*name: str*)
Kill a define!

unget_token(*token*)
Undo token consumption.

class ppci.lang.c.**CParser**(*coptions*, *semantics*)
C Parser.

Implemented in a recursive descent way, like the CLANG[1] frontend for llvm, also without the lexer hack[2]. See for the gcc c parser code [3]. Also interesting is the libfim cparser frontend [4].

The clang parser is very versatile, it can handle C, C++, objective C and also do code completion. This one is very simple, it can only parse C.

[1] http://clang.org/ [2] https://en.wikipedia.org/wiki/The_lexer_hack [3] https://raw.githubusercontent.com/gcc-mirror/gcc/master/gcc/c/ c-parser.c

[4] https://github.com/libfirm/cparser/blob/ 0cc43ed4bb4d475728583eadcf9e9726682a838b/src/parser/parser.c

at_type_id()
Check if the upcoming token is a typedef identifier

is_declaration_statement()
Determine whether we are facing a declaration or not

next_token()
Advance to the next token

parse(*tokens*)
Here the parsing of C is begun ...

Parse the given tokens.

parse_array_designator(*init_cursor*)
Parse array designator like '{2, [10]=4}'

parse_array_string_initializer(*typ*)
Handle the special case where an array is initialized with a string.

parse_asm_operand()
Parse a single asm operand.

parse_asm_operands()
Parse a series of assembly operands. Empty list is allowed.

parse_asm_statement()
Parse an inline assembly statement.

See also: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

parse_attributes()
Parse some attributes.

Examples are:

__attribute__((noreturn))

parse_break_statement()
Parse a break

parse_call(*callee*)
Parse a function call

parse_case_statement()
Parse a case.

For example: 'case 5:'

Or, a gnu extension:

'case 5 ... 10:'

parse_compound_statement()
Parse a series of statements surrounded by '{' and '}'

parse_condition()
Parse an expression between parenthesis

parse_constant_expression()
Parse a constant expression

parse_continue_statement()
Parse a continue statement

parse_decl_group(*decl_spec*)
Parse the rest after the first declaration spec.

For example we have parsed 'static int' and we will now parse the rest. This can be either a function, or a sequence of initializable variables. At least it will probably contain some sort of identifier and an optional pointer stuff.

parse_decl_specifiers(*allow_storage_class=True*)
Parse declaration specifiers.

At the end we know type, storage class and qualifiers.

Gathers storage classes: - typedef - extern - static

One of the following type specifiers: - void, char, int, unsigned, long, ... - typedef-ed type - struct, union or enum

Type qualifiers: - const - volatile

parse_declarations()
Parse normal declarations

parse_declarator(*abstract=False*)
Given a declaration specifier, parse the rest.

This involves parsing optionally pointers and qualifiers and next the declaration itself.

parse_default_statement()
Parse the default case

parse_do_statement()
Parse a do-while statement

parse_empty_statement()
Parse a statement that does nothing!

parse_enum()
Parse an enum definition

parse_enum_fields(*ctyp*, *location*)
Parse enum declarations

parse_expression()
Parse an expression.

See also: http://en.cppreference.com/w/c/language/operator_precedence

parse_for_statement()
Parse a for statement

parse_function_arguments()
Parse function postfix.

We have type and name, now parse function arguments.

parse_function_declaration(*decl_spec*, *declarator*)
Parse a function declaration with implementation

parse_gnu_attribute()
Parse a gnu attribute like __attribute__((noreturn))

parse_goto_statement()
Parse a goto

parse_if_statement()
Parse an if statement

parse_initializer(*typ*)
Parse the C-style array or struct initializer stuff.

Heavily copied from: https://github.com/rui314/8cc/blob/master/parse.c

Argument is the type to initialize.

An initialization can be one of: = 1; = {1, 2, 3}; = {[0] = 3}; // C99 = {.foobar = {23, 3}}; // C99 = {[2..5] = 2}; // C99

parse_initializer_list(*typ*)
Parse braced initializer list.

Parse opening brace, elements and closing brace.

parse_initializer_list_element(*init_cursor*)
Parse an initializer list element with optional designators.

parse_initializer_list_sub(*init_cursor*, *typ*)
Parse braced initializer list.

Parse opening brace, elements and closing brace.

parse_label()
Parse a label statement

parse_primary_expression()
Parse a primary expression

parse_return_statement()
Parse a return statement

parse_statement()
Parse a statement

parse_statement_or_declaration()
Parse either a statement or a declaration

Returns: a list of statements

parse_struct_designator(*init_cursor*)
Parse a struct designator in an initializer list.

parse_struct_fields()
Parse struct or union fields

parse_struct_or_union()
Parse a struct or union

parse_switch_statement()
Parse an switch statement

parse_translation_unit()
Top level start of parsing

parse_type_modifiers(*abstract=False*)
Parse the pointer, name and array or function suffixes.

Can be abstract type, and if so, the type may be nameless.

The result of all this action is a type modifier list with the proper type modifications required by the given code.

parse_typedef(*decl_spec*, *declarator*)
Process typedefs

parse_typename()
Parse a type specifier used in sizeof(int) for example.

parse_variable_declaration(*decl_spec*, *declarator*)
Parse variable declaration optionally followed by initializer.

parse_while_statement()
Parse a while statement

skip_initializer_lists()
Skip superfluous initial values.

class ppci.lang.c.**CAstPrinter**(*file=None*)
Print AST of a C program

visit(*node*)
Recursively visit node's child nodes.

class ppci.lang.c.**CSemantics**(*context*)
This class handles the C semantics

add_statement(*statement*)
Helper to emit a statement into the current block.

apply_type_modifiers(*type_modifiers*, *typ*)
Apply the set of type modifiers to the given type

begin()
Enter a new file / compilation unit.

check_redeclaration_storage_class(*sym*, *declaration*)
Test if we specified an invalid combo of storage class.

coerce(*expr: ppci.lang.c.nodes.expressions.CExpression*, *typ: ppci.lang.c.nodes.types.CType*)
Try to fit the given expression into the given type.

define_tag_type(*tag: str*, *klass: ppci.lang.c.nodes.types.TaggedType*, *location*)
Get or create a tagged type with the given tag kind.

end_function(*body*)
Called at the end of a function

ensure_integer(*expr: ppci.lang.c.nodes.expressions.CExpression*)
Ensure typ is of any integer type.

ensure_no_void_ptr(*expr*)
Test if expr has type void*, and if so, generate a warning and an implicit cast statement.

enter_scope()
Enter a new symbol scope.

equal_types(*typ1*, *typ2*)
Compare two types for equality.

error(*message*, *location*, *hints=None*)
Trigger an error at the given location

finish_compilation_unit()
Called at the end of a file / compilation unit.

get_common_type(*typ1*, *typ2*, *location*)
Given two types, determine the common type.

The common type is a type they can both be cast to.

get_type(*type_specifiers*)
Retrieve a type by type specifiers

init_store(*init_cursor*, *value*)
Store an initial value at position pointed by cursor.

invalid_redeclaration(*sym*, *declaration*, *message='Invalid redefinition'*)
Raise an invalid redeclaration error.

leave_scope()
Leave a symbol scope.

not_impl(*message*, *location*)
Call this function to mark unimplemented code

on_array_designator(*init_cursor*, *index*, *location*)
Handle array designator.

on_array_index(*base*, *index*, *location*)
Check array indexing

on_basic_type(*type_specifiers*, *location*)
Handle basic type

on_binop(*lhs*, *op*, *rhs*, *location*)
Check binary operator

on_builtin_offsetof(*typ*, *member_name*, *location*)
Check offsetof builtin function

on_builtin_va_arg(*arg_pointer*, *typ*, *location*)
Check va_arg builtin function

on_builtin_va_copy(*dest*, *src*, *location*)
Check va_copy builtin function

on_builtin_va_start(*arg_pointer*, *location*)
Check va_start builtin function

on_call(*callee*, *arguments*, *location*)
Check function call for validity

on_case(*value*, *statement*, *location*)
Handle a case statement

on_cast(*to_typ*, *casted_expr*, *location*)
Check explicit casting

on_char(*value*, *location*)
Process a character literal

on_compound_literal(*typ*, *init*, *location*)
Check the consistency of compound literals.

on_default(*statement*, *location*)
Handle a default label

on_do(*body*, *condition*, *location*)
The almost extinct dodo!

on_enum(*tag*, *is_definition*, *location*)
Handle enum declaration

on_enum_value(*ctyp*, *name*, *value*, *location*)
Handle a single enum value definition

on_field_def(*storage_class*, *ctyp*, *name*, *modifiers*, *bitsize*, *location*)
Handle a struct/union field declaration

on_field_designator(*init_cursor*, *field_name*, *location*)
Check field designator.

on_field_select(*base*, *field_name*, *location*)
Check field select expression

on_float(*value*, *location*)
Process floating point literal.

on_for(*initial*, *condition*, *post*, *body*, *location*)
Check for loop construction

on_function_argument(*typ*, *name*, *modifiers*, *location*)
Process a function argument into the proper class

on_function_declaration(*storage_class*, *typ*, *name*, *modifiers*, *location*)
Handle function declaration

on_if(*condition*, *then_statement*, *no*, *location*)
Check if statement

on_number(*value*, *location*)
React on integer numeric literal

on_return(*value*, *location*)
Check return statement

on_sizeof(*typ*, *location*)
Handle sizeof contraption

on_string(*value*, *location*)
React on string literal

on_struct_or_union(*kind*, *tag*, *is_definition*, *fields*, *location*)
Handle struct or union definition.

A definition of a struct occurs when we have: struct S; struct S { int f; }; struct { int f; };

on_switch_exit(*expression*, *statement*, *location*)
Handle switch statement

on_ternop(*lhs*, *op*, *mid*, *rhs*, *location*)
Handle ternary operator 'a ? b : c'

on_type(*typ*, *modifiers*, *location*)
Called when a type itself is described

static on_type_qualifiers(*type_qualifiers*, *ctyp*)
Handle type qualifiers

on_typedef(*typ*, *name*, *modifiers*, *location*)
Handle typedef declaration

on_typename(*name*, *location*)
Handle the case when a typedef is refered

on_unop(*op*, *a*, *location*)
Check unary operator semantics

on_variable_access(*name*, *location*)
Handle variable access

on_variable_declaration(*storage_class*, *typ*, *name*, *modifiers*, *location*)
Given a declaration, and a declarator, create the proper object

on_variable_initialization(*variable*, *expression*)
Handle a variable initialized to some value

on_while(*condition*, *body*, *location*)
Handle the while statement

patch_size_from_initializer(*typ*, *initializer*)
Fill array size from elements!

pointer(*expr*)
Handle several conversions.

Things handled: - Array to pointer decay. - Function to function pointer

promote(*expr*)
Perform integer promotion on expression.

Integer promotion happens when using char and short types in expressions. Those values are promoted to int type before performing the operation.

register_declaration(*declaration*)
Register declaration into the scope.

warning(*message*, *location*, *hints=None*)
Trigger a warning at the given location

class ppci.lang.c.**CSynthesizer**
Take an IR-module and convert it into a C-AST.

This does essentially the opposite of the codegenerator.

syn_block(*block*)
Synthesize an ir block into C

syn_instruction(*instruction*)
Convert ir instruction to its corresponding C counterpart

class ppci.lang.c.**CPrinter**(*f=None*)
Render a C program as text

gen_declaration(*declaration*)
Spit out a declaration

gen_expr(*expr*)
Format an expression as text

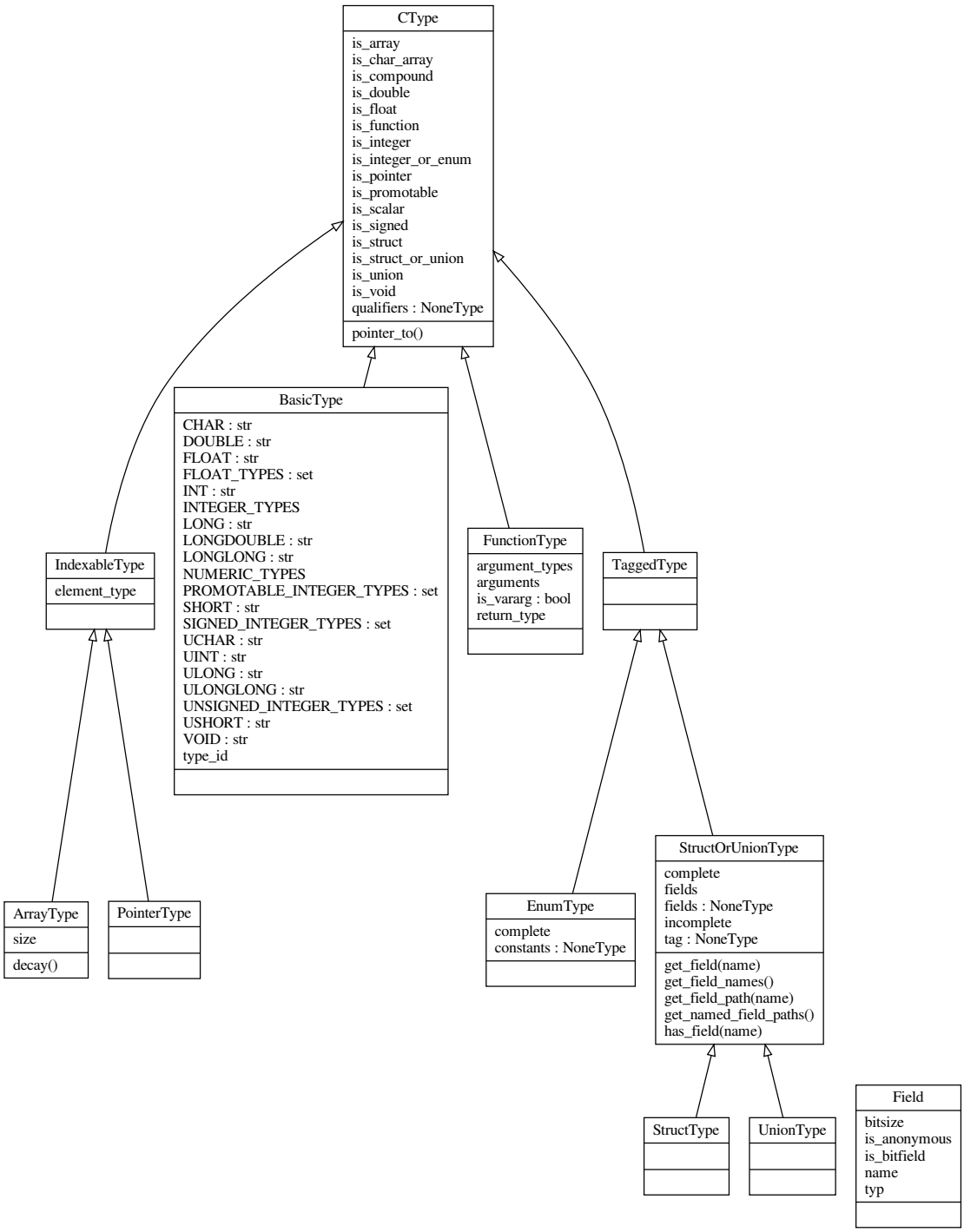
gen_statement(*statement*)
Render a single statement as text

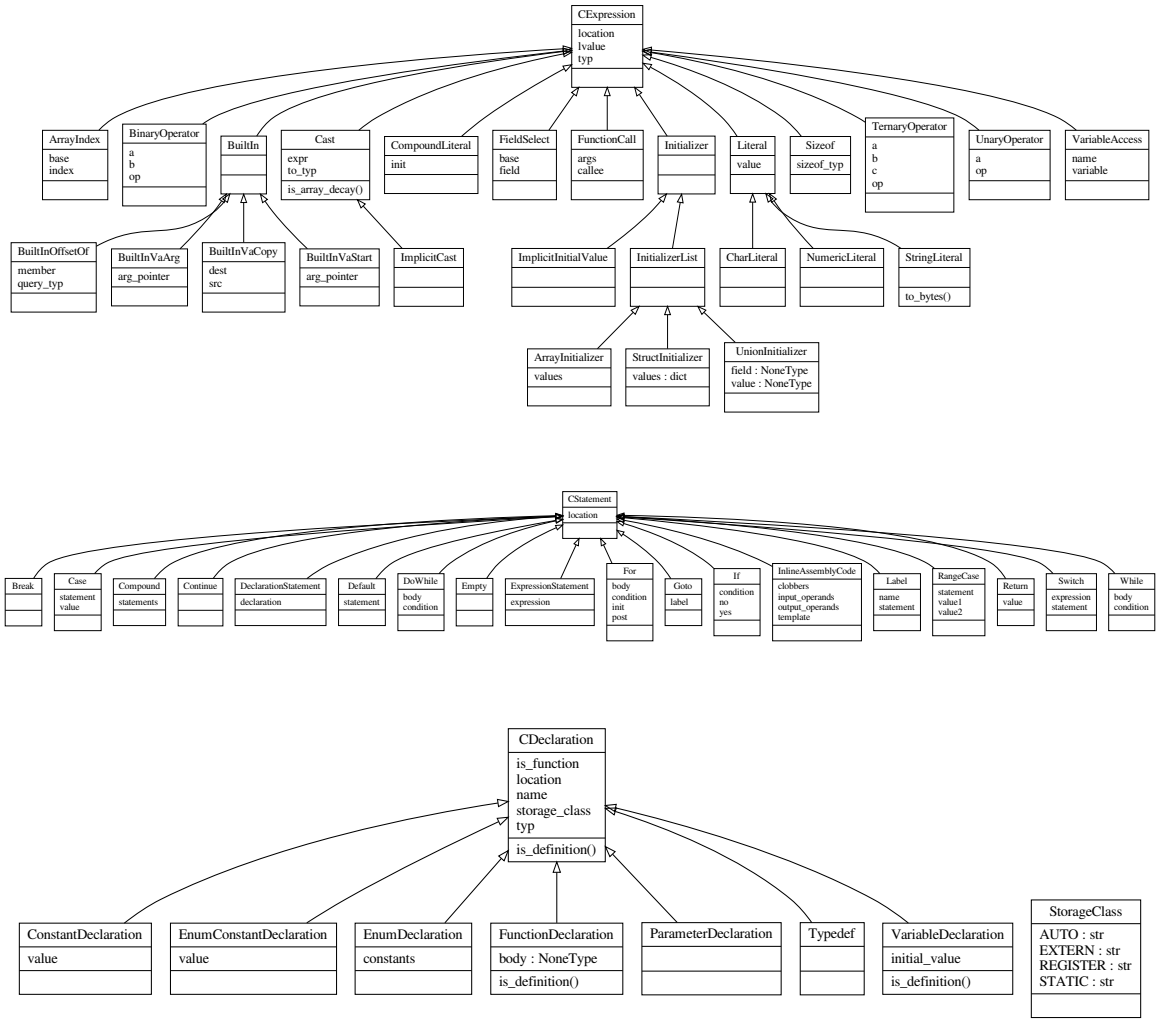
print(*compile_unit*)
Render compilation unit as C

render_type(*typ*, *name=None*)
Generate a proper C-string for the given type

class ppci.lang.c.**CTokenPrinter**
Printer that can turn a stream of token-lines into text

Uml


CType
is_array
is_char_array
is_compound
is_double
is_float
is_function
is_integer
is_integer_or_enum
is_pointer
is_promotable
is_scalar
is_signed
is_struct
is_struct_or_union
is_union
is_void
qualifiers : NoneType
pointer_to()
BasicType
CHAR : str
DOUBLE : str
FLOAT : str
FLOAT_TYPES : set
INT : str
INTEGER_TYPES
LONG : str
LONGDOUBLE : str
LONGLONG : str
NUMERIC_TYPES
PROMOTABLE_INTEGER_TYPES : set
SHORT : str
SIGNED_INTEGER_TYPES : set
UCHAR : str
UINT : str
ULONG : str
ULONGLONG : str
UNSIGNED_INTEGER_TYPES : set
USHORT : str
VOID : str
type_id
IndexableType
element_type
FunctionType
argument_types
arguments
is_vararg : bool
return_type
TaggedType
ArrayType
size
decay()
PointerType
EnumType
complete
constants : NoneType
StructOrUnionType
complete
fields
fields : NoneType
incomplete
tag : NoneType
get_field(name)
get_field_names()
get_field_path(name)
get_named_field_paths()
has_field(name)
StructType
UnionType
Field
bitsize
is_anonymous
is_bitfield
name
typ



References

This section contains some links to other compilers. These were used to draw inspiration from. In the spirit of: it is better to steal ideas then invent something bad yourself :).

Other C compilers

A c99 frontend for libfirm: https://github.com/libfirm/cparser

tcc

tcc: tiny c compiler

This compiler is tiny and uses a single pass to parse and generate code.

lcc

https://github.com/drh/lcc

This compiler does parsing and type checking in one go.

cdt

CDT is an eclipse c/c++ ide.

http://wiki.eclipse.org/CDT/designs/Overview_of_Parsing

Good resources about preprocessors

Cpp internal description:

https://gcc.gnu.org/onlinedocs/cppinternals/

A portable C preprocessor:

http://mcpp.sourceforge.net

The boost wave preprocessor:

http://www.boost.org/doc/libs/1_63_0/libs/wave/

A java implementation of the preprocessor:

http://www.anarres.org/projects/jcpp/

https://github.com/shevek/jcpp

CDT preprocessor: http://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/parser/scanner/CPreprocessor.java

The lcc preprocessor part:

https://github.com/drh/lcc/blob/master/cpp/cpp.h

3.4.5 Fortran

The *ppci.lang.fortran* module contains functionality to deal with fortran code.

Warning: This module is a work in progress.

Module

This is the fortran frontend.

Currently this front-end is a work in progress.

ppci.lang.fortran.**fortran_to_ir**(*source*)
Translate fortran source into IR-code

class ppci.lang.fortran.**FortranParser**
Parse some fortran language

parse(*src*)
parse a piece of FORTRAN77

parse_assignment()
Parse an assignment

parse_declaration()
Parse variable declarations

parse_end()
Parse end statement

parse_expression(*bp=0*)
Welcome to expression parsing!

Solve this using precedence parsing with binding power.

parse_fmt_spec()
Parse a format specifier

parse_for()
Parse a for loop statement

parse_format()
Parse a format statement

parse_go()
Parse go to syntax

parse_label_ref()
Parse a label reference, this can be a number or.. a variable?

parse_program()
Parse a program

parse_unit_spec()
Parse a unit (file) specifier

class ppci.lang.fortran.**Visitor**
Visit ast nodes

class ppci.lang.fortran.**Printer**

print(*node*)
Print the AST

3.4.6 Java

Warning: This module is a work in progress.

Java is perhaps the most used programming language in the world.

PPCI offers some functions to deal with compiled Java bytecode (known as .class files) and archives of multiple .class files (.jar files).

Compile Java ahead of time

It's possible to compile a subset of Java into machine code. Say, we have some Java code:

```
class Test14 {
    static int my_add(int a, int b) {
        return a + b + 1;
    }
}
```

We can compile this with `javac`, and next up, compile it with PPCI into msp430 code:

```
$ javac Test14.java
$ python -m ppci.cli.java compile Test14.class -m msp430
```

Load a class file dynamically

Given that you created a class file with a static function my_add in it (using `javac`), you could do the following:

```
>>> from ppci.arch.jvm import load_class
>>> klass = load_class('add.class')
>>> klass.my_add(1, 5)
7
```

This example is located in the file examples/java/load.py

Links to similar projects

- pyjvm https://github.com/andrewromanenco/pyjvm
- pyjvm https://github.com/ronyhe/pyjvm
- jawa https://github.com/TkTech/Jawa

Module reference

Java virtual machine (JVM).

This module supports loading and saving of java bytecode.

See also:

https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

ppci.arch.jvm.**class_to_ir**(*class_file*)
Translate java class file into IR-code.

ppci.arch.jvm.**read_class_file**(*f*, *verbose=False*)
Read a class file.

ppci.arch.jvm.**read_jar**(*filename*)
Take a stroll through a java jar file.

ppci.arch.jvm.**load_class**(*filename*)
Load a compiled class into memory.

ppci.arch.jvm.**print_class_file**(*class_file*)
Dump a class file.

Module to load class/jar files.

Another really good python java package: https://github.com/TkTech/Jawa http://jawa.tkte.ch/

class ppci.arch.jvm.io.**DescriptorParser**(*text*)
Descriptor string parser.

parse_method_descriptor()
Parse a method descriptor.

class ppci.arch.jvm.io.**JavaFileReader**(*f*, *verbose=False*)
Java class file reader.

read_attribute_info()
Read a single attribute.

read_attributes()
Read a series of attributes.

read_class_file()
Read a class file.

read_constant_pool()
Read the constant pool.

read_constant_pool_info()
Read a single tag from the constant pool.

read_field_info()
Read field info structure.

read_fields()
Read the fields of a class file.

read_flags()
Process flag field.

read_interfaces()
Read all interfaces from a class file.

read_method_info()
Read method info structure

read_methods()
Read the methods from a classfile.

class ppci.arch.jvm.io.**JavaFileWriter**
Enables writing of java class files.

ppci.arch.jvm.io.**disassemble**(*bytecode*)
Process a bytecode slab into instructions.

ppci.arch.jvm.io.**read_class_file**(*f*, *verbose=False*)
Read a class file.

ppci.arch.jvm.io.**read_jar**(*filename*)
Take a stroll through a java jar file.

ppci.arch.jvm.io.**read_manifest**(*f*)
Read a jarfile manifest.

3.4.7 Llvm

Front-end for the LLVM IR-code

This front-end can be used as an enabler for many other languages, for example ADA and C++.

Currently this module a work in progress. The first step is to parse the llvm assembly. The next phase would be to convert that into ppci ir.

Another nice idea is to generate llvm ir code from ppci. When generating and parsing are combined, the llvm optimizers can be used.

ppci.lang.llvmir.**llvm_to_ir**(*source*)
Convert llvm assembly code into an IR-module

Example usage

An example usage is the following:

```
$ llvm-stress | ppci-llc.py -m riscv -o my_object.oj -
```

Here, the llvm-stress tool generates some random llvm source code, and this is piped into the ppci-llc.py command, which takes ll code and turns it into machine code.

Another example is how to use clang together with ppci-llc.py:

```
$ clang -S -emit-llvm -o - magic2.c | ppci-llc.py -o my_obj.oj -m msp430 -
```

This will compile the sourcefile magic2.c into my_obj.oj for the msp430 target.

3.4.8 OCaml

Warning: This module is a work in progress.

OCaml is a functional programming language. Its output is bytecode.

This is an exploration on how to compile this bytecode.

Produce bytecode for example by:

```
$ ocamlc -o test.byte test.ml
$ python -m ppci.cli.ocaml disassemble test.byte
$ FUTURE: python -m ppci.cli.ocaml opt test.byte -m m68k
```

Module

ppci.lang.ocaml.**read_file**(*filename*)
Read a cmo or bytecode file.

ppci.lang.ocaml.**ocaml_to_ir**(*module*)
Transform ocaml bytecode into ir-code.

3.4.9 Pascal

The *ppci.lang.pascal* module contains functionality to transform pascal code into IR-code.

Warning: This module is a work in progress.

Module

Pascal front-end

class ppci.lang.pascal.**PascalBuilder**(*diag*, *arch_info*)
Generates IR-code from pascal source.

build(*sources*)
Build the given sources.

Raises compiler error when something goes wrong.

do_parse(*src*, *context*)
Lexing and parsing stage (phase 1)

class ppci.lang.pascal.**Parser**(*diag*)
Parses pascal into ast-nodes

add_symbol(*sym*)
Add a symbol to the current scope

do_coerce(*expr: ppci.lang.pascal.nodes.expressions.Expression*, *to_type: ppci.lang.pascal.nodes.types.Type*)
Try to convert expression into the given type.

expr: the expression value with a certain type typ: the type that it must be Raises an error is the conversion cannot be done.

enter_scope()
Enter a lexical scope.

get_common_type(*a*, *b*, *loc*)
Determine the greatest common type.

This is used for coercing binary operators. For example:

- int + float -> float
- byte + int -> int

- byte + byte -> byte
- pointer to x + int -> pointer to x

leave_scope()
Leave the current lexical scope.

parse_actual_parameter_list(*parameter_types*)
Parse a list of parameters

parse_binop_with_precedence(*priority*) → ppci.lang.pascal.nodes.expressions.Expression
Parse binary operators using a binding strength.

This is a neat trick to parse expressions without a whole bunch of similar looking functions for each operator. We use the concept of binding strength, or priority to group expressions according to operator precendence.

parse_block()
Parse a block.

A block being constants, types, variables and statements.

parse_builtin_procedure_call(*func: str*, *location*)
Do sort of macro expansion of built-in procedure call.

parse_case_of() → ppci.lang.pascal.nodes.statements.CaseOf
Parse case-of statement

parse_compound_statement()
Parse a compound statement

parse_constant_definitions()
Parse constant definitions.

This has the form:

'const' 'ID' '=' expr; 'ID' '=' expr;

parse_designator()
A designator designates an object with a name.

parse_enum_type_definition()
Parse enumerated type definition.

This looks like:

colors = (red, green, blue)

parse_expression() → ppci.lang.pascal.nodes.expressions.Expression
Parse a an expression.

parse_expression_list()
Parse one or more expressions seperated by ','

parse_for() → ppci.lang.pascal.nodes.statements.For
Parse a for statement

parse_formal_parameter_list()
Parse format parameters to a subroutine.

These can be immutable values, variables, or function pointers.

parse_function_declarations()
Parse all upcoming function / procedure definitions

parse_function_def()
Parse function definition

parse_id_sequence()
Parse one or more identifiers seperated by ','

parse_if_statement()
Parse if statement

parse_one_or_more(*parse_function*, *seperator: str*)
Parse one or more occurences parsed by parse_function seperated by seperator.

parse_primary_expression() → ppci.lang.pascal.nodes.expressions.Expression
Literal and parenthesis expression parsing

parse_procedure_call(*symbol*, *location*)
Procedure call.

This can be either a builtin procedure, or a user defined procedure. Builtin procedure are somewhat magical in that they are sort-of-macro-expanded at compile time.

parse_program(*context*)
Parse a program

parse_record_fixed_list()
Parse fixed parts of a record type definition.

parse_record_type_definition(*packed*)
Parse record type description.

parse_record_variant()
Parse case .. of part.

parse_repeat()
Parses a repeat statement

parse_return() → ppci.lang.pascal.nodes.statements.Return
Parse a return statement

parse_single_variable_declaration()
Parse a single variable declaration line ending in ';'

parse_single_with_variable()
Parse a single with statement variable.

parse_source(*tokens*, *context*)
Parse a module from tokens

parse_statement() → ppci.lang.pascal.nodes.statements.Statement
Determine statement type based on the pending token

parse_type_definitions()
Parse type definitions.

These have the form:

'type' 'ID' '=' type-spec ';' 'ID' '=' type-spec ';' ...

parse_type_spec(*packed=False*)
Parse type specification.

This can be any type, from record to ordinal or boolean.

parse_uses()
Parse import construct

parse_variable()
Parse access to a variable with eventual accessor suffixes.

parse_variable_access(*symbol*, *location*)
Process any trailing variable access.

parse_variable_declarations()
Parse variable declarations

parse_while() → ppci.lang.pascal.nodes.statements.While
Parses a while statement

require_boolean(*expr*)
Check the type of expression to be boolean, and raise an error if not.

class ppci.lang.pascal.**Lexer**(*diag*)
Generates a sequence of token from an input stream

tokenize(*text*)
Keeps track of the long comments

3.4.10 Python compilation

The ppci.lang.python module can be used to translate python code into wasm, ir or machine code.

ppci.lang.python.**python_to_ir**(*f*, *imports=None*)
Compile a piece of python code to an ir module.

Parameters

- **f** (*file-like-object*) – a file like object containing the python code
- **imports** – Dictionary with symbols that are present.

Returns A *ppci.ir.Module* module

```
>>> import io
>>> from ppci.lang.python import python_to_ir
>>> f = io.StringIO("def calc(x: int) -> int: return x + 133")
>>> python_to_ir(f)
<ppci.ir.Module object at ...>
```

ppci.lang.python.**ir_to_python**(*ir_modules*, *f*, *reporter=None*)
Convert ir-code to python code

ppci.lang.python.**jit**(*function*)
Jitting function decorator.

Can be used to just-in-time (jit) compile and load a function. When a function is decorated with this decorator, the python code is translated into machine code and this code is loaded in the current process.

For example:

```
from ppci.lang.python import jit

@jit
def heavymath(a: int, b: int) -> int:
    return a + b
```

Now the function can be used as before:

```
>>> heavymath(2, 7)
9
```

ppci.lang.python.**load_py**(*f*, *imports=None*, *reporter=None*)
Load a type annotated python file.

Parameters **f** – a file like object containing the python source code.

ppci.lang.python.**python_to_wasm**(**sources*)
Compile Python functions to wasm, by using Python's ast parser and compiling a very specific subset to WASM instructions. All values are float64. Each source can be a string, a function, or AST.

3.4.11 S-expressions

The ppci.lang.sexpr module can be used to read S-expressions.

```
>>> from ppci.lang.sexpr import parse_sexpr
>>> parse_sexpr('(bla 29 ("hello world" 1337 @#!))')
('bla', 29, ('hello world', 1337, '@#!'))
```

Reference

Functionality to tokenize and parse S-expressions.

ppci.lang.sexpr.**parse_sexpr**(*text: str*, *multiple=False*) → tuple
Parse S-expression given as string. Returns a tuple that represents the S-expression.

3.4.12 Language tools

The ppci.lang.tools package contains tools for programming language construction.

Diagnostics

To provide useful feedback about programming errors, there are classes for sourcecode location and diagnostics.

class ppci.lang.common.**SourceLocation**(*filename*, *row*, *col*, *ln*, *source=None*)
A location that refers to a position in a source file

get_source_line()
Return the source line indicated by this location

print_message(*message*, *lines=None*, *filename=None*, *file=None*)
Print a message at this location in the given source lines

class ppci.lang.common.**SourceRange**(*p1*, *p2*)

p1
Alias for field number 0

p2
Alias for field number 1

class ppci.lang.common.**Token**(*typ*, *val*, *loc*)
Token is used in the lexical analyzer. The lexical analyzer takes a text and splits it into tokens.

ppci.lang.common.**print_line**(*row*, *lines*, *file=None*)
Print a single source line

Lexing

class ppci.lang.tools.baselex.**BaseLexer**(*tok_spec*)
Base class for a lexer.

This class can be overridden to create a lexer. This class handles the regular expression generation and source position accounting.

feed(*txt*)
Feeds the lexer with extra input

newline()
Enters a new line

tokenize(*txt*, *eof=False*)
Generator that generates lexical tokens from text.

Optionally yield the EOF token.

class ppci.lang.tools.baselex.**LexMeta**
Meta class which inspects the functions decorated with 'on'

class ppci.lang.tools.baselex.**SimpleLexer**
Simple class for lexing.

Use this class by subclassing it and decorating handler methods with the 'on' function.

gettok()
Find a match at the given position

tokenize(*txt*, *eof=False*)
Generator that generates lexical tokens from text.

Optionally yield the EOF token.

ppci.lang.tools.baselex.**on**(*pattern*, *flags=0*, *order=0*)
Register method to the given pattern.

Parameters **order** – a sorting priority. Lower number comes first.

Grammar

To help to define a grammar for a language the grammar classes can be used to define a language grammar. From this grammar parsers can be generated.

A grammar can be defined like this:

```
>>> from ppci.lang.tools.grammar import Grammar
>>> g = Grammar()
>>> g.add_terminal('term')
>>> g.add_production('expr', ['expr', '+', 'term'])
>>> g.add_production('expr', ['expr', '-', 'term'])
>>> g.dump()
Grammar with 2 rules and 1 terminals
expr -> ('expr', '+', 'term') P_0
expr -> ('expr', '-', 'term') P_0
>>> g.is_terminal('expr')
False
>>> g.is_terminal('term')
True
```

class ppci.lang.tools.grammar.**Grammar**
Defines a grammar of a language

add_one_or_more(*element_nonterm*, *list_nonterm*)
Helper to add the rule lst: elem lst: lst elem

add_production(*name*, *symbols*, *semantics=None*, *priority=0*)
Add a production rule to the grammar

add_terminal(*name*)
Add a terminal name

add_terminals(*terminals*)
Add all terminals to terminals for this grammar

check_symbols()
Checks no symbols are undefined

create_combinations(*rule*, *non_terminal*)
Create n copies of rule where nt is removed. For example replace: A -> B C B by: A -> B C B A -> B C A -> C B A -> C if: B -> epsilon

dump()
Print this grammar

is_nonterminal(*name*)
Check if a name is a non-terminal

is_normal
Check if this grammar is normal. Which means: - No empty productions (epsilon productions)

is_terminal(*name*)
Check if a name is a terminal

productions_for_name(*name*)
Retrieve all productions for a non terminal

rewrite_eps_productions()
Make the grammar free of empty productions. Do this by permutating all combinations of rules that would otherwise contain an empty place.

symbols
Get all the symbols defined by this grammar

class ppci.lang.tools.grammar.**Production**(*name*, *symbols*, *semantics*, *priority=0*)
Production rule for a grammar. It consists of a left hand side non-terminal and a list of symbols as right hand side. Also it contains a function that must be called when this rule is applied. The right hand side may contain terminals and non-terminals.

is_epsilon
Checks if this rule is an epsilon rule

ppci.lang.tools.grammar.**print_grammar**(*g*)
Pretty print a grammar

Earley parser

Implementation of the earley parser strategy.

See also: - https://en.wikipedia.org/wiki/Earley_parser

And the book: Parsing Techniques: A Practical Guide (2nd edition)

class ppci.lang.tools.earley.**Column**(*i*, *token*)
A set of partially parsed items for a given token position

class ppci.lang.tools.earley.**EarleyParser**(*grammar*)
Earley parser.

As opposed to an LR parser, the Earley parser does not construct tables from a grammar. It uses the grammar when parsing.

The Earley parser has 3 key functions:

- predict: what have we parsed so far, and what productions can be made with this.
- scan: parse the next input symbol, en create a new set of possible parsings
- complete: when we have scanned something according to a rule, this rule can be applied.

When an earley parse is complete, the parse can be back-tracked to yield the resulting parse tree or the syntax tree.

complete(*completed_item*, *start_col*, *current_column*)
Complete a rule, check if any other rules can be shifted!

make_tree(*columns*, *nt*)
Make a parse tree

parse(*tokens*, *debug_dump=False*)
Parse the given token string

predict(*item*, *col*)
Add all rules for a certain non-terminal

scan(*item*, *col*)
Check if the item can be shifted into the next column

walk(*columns*, *end*, *nt*)
Process the parsed columns back to a parse tree

class ppci.lang.tools.earley.**Item**(*rule*, *dot*, *origin*)
Partially parsed grammar rule

Recursive descent

Recursive descent parsing (also known as handwritten parsing) is an easy way to parse sourcecode.

class ppci.lang.tools.recursivedescent.**RecursiveDescentParser**
Base class for recursive descent parsers

consume(*typ=None*)
Assert that the next token is typ, and if so, return it.

If typ is a list or tuple, consume one of the given types. If typ is not given, consume the next token.

error(*msg*, *loc=None*)
Raise an error at the current location

has_consumed(*typ*)
Checks if the look-ahead token is of type typ, and if so eats the token and returns true

init_lexer(*tokens*)
Initialize the parser with the given tokens (an iterator)

look_ahead(*amount*)
Take a look at x tokens ahead

next_token()
Advance to the next token

not_impl(*msg=''*)
Call this function when parsing reaches unimplemented parts

peek
Look at the next token to parse without popping it

LR parsing

class ppci.lang.tools.lr.**Item**(*production*, *dotpos*, *look_ahead*)
Represents a partially parsed item It has a production it is looking for, a position in this production called the 'dot' and a look ahead symbol that must follow this item.

NextNext
Gets the symbol after the next symbol, or EPS if at the end

can_shift_over(*symbol*)
Determines if this item can shift over the given symbol

is_reduce
Check if this item has the dot at the end

shifted()
Creates a new item that is shifted one position

class ppci.lang.tools.lr.**LrParser**(*grammar*, *action_table*, *goto_table*)
LR parser automata. This class takes goto and action table and can then process a sequence of tokens.

parse(*lexer*)
Parse an iterable with tokens

class ppci.lang.tools.lr.**LrParserBuilder**(*grammar*)
Construct goto and action tables according to LALR algorithm

closure(*itemset*)
Expand itemset by using epsilon moves

first
The first set is a mapping from a grammar symbol to a set of set of all terminal symbols that can be the first terminal when looking for the grammar symbol

gen_canonical_set(*iis*)
Create all LR1 states

generate_parser()
Generates a parser from the grammar

generate_tables()
Generate parsing tables

initial_item_set()
Calculates the initial item set

next_item_set(*itemset*, *symbol*)
Determines the next itemset for the current set and a symbol This is the goto procedure

class ppci.lang.tools.lr.**Reduce**(*rule*)
Reduce according to the given rule

class ppci.lang.tools.lr.**Shift**(*to_state*)
Shift over the next token and go to the given state

class ppci.lang.tools.lr.**State**(*items*, *number*)
A state in the parsing machine. A state contains a set of items and a state number

ppci.lang.tools.lr.**calculate_first_sets**(*grammar*)
Calculate first sets for each grammar symbol This is a dictionary which maps each grammar symbol to a set of terminals that can be encountered first when looking for the symbol.

Parser generator script

class ppci.lang.tools.yacc.**XaccGenerator**
Generator that writes generated parser to file

generate_python_script()
Generate python script with the parser table

print(**args*)
Print helper function that prints to output file

class ppci.lang.tools.yacc.**XaccLexer**

tokenize(*txt*)
Generator that generates lexical tokens from text.

Optionally yield the EOF token.

class ppci.lang.tools.yacc.**XaccParser**
Implements a recursive descent parser to parse grammar rules.

We could have made an generated parser, but that would yield a chicken egg issue.

parse_grammar(*tokens*)
Entry parse function into recursive descent parser

parse_rhs()
Parse the right hand side of a rule definition

parse_rule()
Parse a rule definition

ppci.lang.tools.yacc.**load_as_module**(*filename*)
Load a parser spec file, generate LR tables and create module

3.5 Binary utilities

This section describes the various binary utility functions.

3.5.1 Linker

The linker module implements the various tasks of linking. This includes:

- Symbol resolution
- Applying relocations
- Section merging
- Layout of section into memory
- Linker relaxation

Module reference

Linker utility.

class ppci.binutils.linker.**Linker**(*arch*, *reporter=None*)
Merges the sections of several object files and performs relocation

add_missing_symbols_from_libraries(*libraries*)
Try to fetch extra code from libraries to resolve symbols.

Note that this can be a rabbit hole, since libraries can have undefined symbols as well.

check_undefined_symbols()
Find undefined symbols.

do_relaxations()
Linker relaxation. Just relax ;).

Linker relaxation is the process of finding shorted opcodes for jumps to addresses nearby.

For example, an instruction set might define two jump operations. One with a 32 bits offset, and one with an 8 bits offset. Most likely the compiler will generate conservative code, so always 32 bits branches. During the relaxation phase, the code is scanned for possible replacements of the 32 bits jump by an 8 bit jump.

Possible issues that might occur during this phase:

- alignment of code. Code that was previously aligned might be shifted.
- Linker relaxations might cause the opposite effect on jumps whose distance increases due to relaxation. This occurs when jumping over a memory whole between sections.

do_relocations()
Perform the correct relocation as listed

get_symbol_value(*symbol_id*)
Get value of a symbol from object or fallback

get_undefined_symbols()
Get a list of currently undefined symbols.

inject_object(*obj*, *debug*)
Paste object into destination object.

inject_symbol(*name*, *binding*, *section*, *value*, *typ*, *size*)
Generate new symbol into object file.

layout_sections(*layout*)
Use the given layout to place sections into memories

link(*input_objects*, *layout=None*, *partial_link=False*, *debug=False*, *extra_symbols=None*, *libraries=None*, *entry_symbol_name=None*)
Link together the given object files using the layout

merge_global_symbol(*name*, *section*, *value*, *typ*, *size*)
Insert or merge a global name.

merge_objects(*input_objects*, *debug*)
Merge object files into a single object file

report_link_result()
After linking is complete, this function can be used to dump information to a reporter.

ppci.binutils.linker.**link**(*objects*, *layout=None*, *use_runtime=False*, *partial_link=False*, *reporter=None*, *debug=False*, *extra_symbols=None*, *libraries=None*, *entry=None*)
Links the iterable of objects into one using the given layout.

Parameters

- **objects** – a collection of objects to be linked together.
- **layout** – optional memory layout.
- **use_runtime** (*bool*) – also link compiler runtime functions
- **partial_link** – Set this to true if you want to perform a partial link. This means, undefined symbols are no error.
- **debug** (*bool*) – when true, keep debug information. Otherwise remove this debug information from the result.
- **extra_symbols** – a dict of extra symbols which can be used during linking.
- **libraries** – a list of libraries to use when searching for symbols.
- **entry** – the entry symbol where execution should begin.

Returns The linked object file

```
>>> import io
>>> from ppci.api import asm, c3c, link
>>> asm_source = io.StringIO("db 0x77")
>>> obj1 = asm(asm_source, 'arm')
>>> c3_source = io.StringIO("module main; var int a;")
>>> obj2 = c3c([c3_source], [], 'arm')
>>> obj = link([obj1, obj2])
>>> print(obj)
CodeObject of 8 bytes
```

3.5.2 Object archiver

The object archiver has similar function to the GNU ar utility. Essentially an archive is a zip with object files.

Module reference

Grouping of multiple object files into a single archive.

class ppci.binutils.archive.**Archive**(*objs*)
The archive. Holder of object files. Similar to GNU ar.

classmethod load(*f*)
Load archive from disk.

save(*output_file*)
Save archive to file.

ppci.binutils.archive.**archive**(*objs*)
Create an archive from multiple object files.

ppci.binutils.archive.**get_archive**(*filename*)
Load an archive from file.

3.5.3 Memory layout

The memory layout can be used to define memory layout. It is similar to a linker script, but less flexible.

Module reference

class ppci.binutils.layout.**Align**(*alignment*)
Align the current position to the given byte

class ppci.binutils.layout.**EntrySymbol**(*symbol_name*)
Specify the entry symbol of this file.

class ppci.binutils.layout.**Layout**
Defines a layout for the linker to be used

static load(*file*)
Load a layout from file

class ppci.binutils.layout.**LayoutLexer**
Lexer for layout files

class ppci.binutils.layout.**Memory**(*name*)
Specification of how a memory may look like and what it contains.

class ppci.binutils.layout.**Section**(*section_name*)
Insert a section here

class ppci.binutils.layout.**SectionData**(*section_name*)
Insert only the data of a section here, not the section itself

ppci.binutils.layout.**get_layout**(*layout*)
Get a layout from object or file

3.5.4 Object format

Compiled code is stored in a different format then the usual ELF format. A new format is used, coined 'oj'. It is valid json and contains all the information that is also in ELF, but then in more plain text format. You can open and even edit an oj-object file with a text editor.

Object files are used to store assembled code.

Information contained is code, symbol table and relocation information.

The hierarchy is as follows:

- an object file contains zero or more sections.
- an object file may contains memory definitions, which contain a sequence of contained sections.
- a section contains a name and data.
- relocations are offset into a section, refer a symbol and have a type
- symbols are offset into a section and have a name
- debug data have an offset into a section and contain data.
- sections cannot overlap

class ppci.binutils.objectfile.**Image**(*name: str*, *address: int*)
Memory image.

A memory image is a piece that can be loaded into memory.

add_section(*section*)
Add a section to this memory image

data
Get the data of this memory

size
Determine the size of this memory

class ppci.binutils.objectfile.**ObjectFile**(*arch*)
Container for sections with compiled code or data. Also contains symbols and relocation entries. Also contains debug information.

add_image(*image*)
Add an image

add_relocation(*reloc*)
Add a relocation entry

add_section(*section*)
Add a section

add_symbol(*id*, *name*, *binding*, *value*, *section*, *typ*, *size*)
Define a new symbol

byte_size
Get the size in bytes of this object file

create_section(*name*)
Create and add a section with the given name.

del_symbol(*name*)
Remove a symbol with a given name

get_defined_symbols()
Get a list of defined symbols.

get_image(*name*)
Get a memory image

get_section(*name*, *create=False*)
Get or create a section with the given name

get_symbol(*name*)
Get a symbol

get_symbol_id_value(*symbol_id*)
Lookup a symbol and determine its value

get_undefined_symbols()
Get a list of undefined symbols.

has_section(*name*)
Check if the object file has a section with the given name

has_symbol(*name*)
Check if this object file has a symbol with name 'name'

is_executable
Test if this object file is executable by checking the entry point.

static load(*input_file*)
Load object file from file

save(*output_file*)
Save object file to a file like object

serialize()
Serialize the object into a dictionary structure suitable for json.

class ppci.binutils.objectfile.**RelocationEntry**(*reloc_type*, *symbol_id*, *section*, *offset*, *addend*)
A relocation entry.

While it mich be confusing to have Relocation here, and RelocationType in arch.encoding, this is cleaner, since the relocation here is a record indicating a relocation, while the relocation type is emitted by instructions and can actually apply the relocation.

This class is a record holding information on where to a apply which relocation. Not how to do the actual relocation.

class ppci.binutils.objectfile.**Section**(*name*)
A defined region of data in the object file

add_data(*data*)
Append data to the end of this section

class ppci.binutils.objectfile.**Symbol**(*id*, *name*, *binding*, *value*, *section*, *typ*, *size*)
A symbol definition in an object file

is_function
Test if this symbol is a function.

undefined
Test if this symbol is undefined.

ppci.binutils.objectfile.**deserialize**(*data*)
Create an object file from dict-like data

ppci.binutils.objectfile.**get_object**(*obj*)
Try hard to load an object

ppci.binutils.objectfile.**merge_memories**(*mem1*, *mem2*, *name*)
Merge two memories into a new one

ppci.binutils.objectfile.**print_object**(*obj*)
Display an object in a user friendly manner

ppci.binutils.objectfile.**serialize**(*x*)
Serialize an object so it can be json-ified, or serialized

3.6 Build system

It can be convenient to bundle a series of build steps into a script, for example a makefile. Instead of depending on make, yet another build tool was created. The build specification is specified in xml. Much like msbuild and Ant.

A project can contain a build.xml file which describes how the project should be build. The name of the file can be build.xml or another filename. This file can than be given to *ppci-build*.

An example build file:

```
<project name="Snake" default="snake">
    <import name="ppci.build.buildtasks" />

    <target name="snake">
        <assemble
            source="../startup.asm"
            arch="arm:thumb"
            output="startup.oj" />
        <c3compile
            arch="arm:thumb"
            sources="../../src/snake/*.c3;../bsp.c3;../../../librt/io.c3"
            debug="true"
            output="rest.oj"
            report="snake_report.html"/>
        <link output="snake.oj"
            layout="../memlayout.mmap"
            debug="true"
            objects="startup.oj;rest.oj" />
        <objcopy
            objectfile="snake.oj"
            imagename="flash"
            format="bin"
            output="snake.bin" />
    </target>

</project>
```

3.6.1 Projects

The root element of a build file is the project tag. This tag contains a name and optionally a default target attribute. When no target is given when building the project, the default target is selected.

3.6.2 Targets

Like make, targets can depend on eachother. Then one target is run, the build system makes sure to run depending targets first. Target elements contain a list of tasks to perform.

3.6.3 Tasks

The task elements are contained within target elements. Each task specifies a build action. For example the link task takes multiple object files and combines those into a merged object.

3.7 IR

3.7.1 IR-code

The purpose of an intermediate representation (IR) of a program is to decouple the implementation of a front-end from the implementation of a back-end. That is, front ends generate IR-code, optimizers optimize this code and lastly backends transform it into machine code or something else.

A good IR has several characteristics:

- It should be simple enough for front-ends to generate code.
- It should be rich enough to exploit the target instructions best.

The IR in the ppci.ir module has the following properties:

- It is static single assignment form. Meaning a value can only be assigned once, and is then never changed. This has several advantages.
- It contains only basic types. Structures, arrays and void types are not represented.

Top level structure

The IR-code is implemented in the ir package.

class ppci.ir.**Module**(*name: str*, *debug_db=None*)
Container unit for variables and functions.

add_external(*external*)
Add an externally located thing

add_function(*function*)
Add a function to this module

add_variable(*variable*)
Add a variable to this module

display()
Display this module

functions
Get all functions of this module

get_function(*name: str*)
Get a function with the given name

stats()
Returns a string with statistic information such as block count

variables
Get all variables of this module

External declarations

class ppci.ir.**External**(*name*)
External object

class ppci.ir.**ExternalSubRoutine**(*name*, *argument_types*)
External subroutine base class

class ppci.ir.**ExternalProcedure**(*name*, *argument_types*)
External procedure

class ppci.ir.**ExternalFunction**(*name*, *argument_types*, *return_ty*)
External function

Global declarations

class ppci.ir.**Binding**
Enum for public / private-ness of global values.

This can be used to keep value local to the module. This can be useful when you compile two modules with symbols with the same name. If the symbols are defined as local, this will not cause a name clash during linking.

class ppci.ir.**Variable**(*name*, *binding*, *amount*, *alignment*, *value=None*)
Global variable, reserves room in the data area. Has name and size

class ppci.ir.**SubRoutine**(*name*, *binding*)
Base class of function and procedure. These two differ in that a function returns a value, where as a procedure does not.

A subroutine contains basic blocks which refer to each other, forming a control flow graph (CFG). Each SubRoutine has a single entry basic block.

Design trade-off: In C, a void type is introduced to permit functions that return nothing (void). This seems somewhat artificial, but keeps things simple for the users. In pascal, the procedure and function types are explicit, and the void type is not needed. This is also the approach taken here.

So instead of a Function and Call types, we have *Function*, *Procedure*, *FunctionCall* and *ProcedureCall* types.

add_block(*block*)
Add a block to this function.

Parameters **block** (*Block*) – the basic block to add.

add_parameter(*parameter*)
Add an argument to this function

block_names
Get the names of all the blocks in this function

calc_reachable_blocks()
Determine all blocks that can be reached

delete_unreachable()
Calculate all reachable blocks from entry and delete all others

dump()
Print this function

get_instructions()
Get all instructions in this routine.

get_out_calls()
Return the calls that leave this function.

is_function
Test if this routine is a function.

is_leaf()
Test if this procedure is a leaf function.

A leaf function calls no other functions.

is_procedure
Test if this routine is a procedure.

make_unique_name(*dut*)
Check if the name of the given dut is unique and if not make it so. Also add it to the used names

num_instructions()
Count the number of instructions contained in this function

remove_block(*block*)
Remove a block from this function

class ppci.ir.**Procedure**(*name*, *binding*)
A procedure definition that does not return a value.

Parameters

- **name** (*str*) – the name of the procedure
- **binding** (*Binding*) – The linkage of this procedure

class ppci.ir.**Function**(*name*, *binding*, *return_ty*)
Represents a function.

A function always returns a value.

Parameters

- **name** (*str*) – the name of the procedure
- **binding** (*Binding*) – The linkage of this procedure
- **return_ty** – The return value of this function.

Basic block

class ppci.ir.**Block**(*name*)
Uninterrupted sequence of instructions.

A block is properly terminated if its last instruction is a *FinalInstruction*.

add_instruction(*instruction*)
Add an instruction to the end of this block

change_target(*old*, *new*)
Change the target of this block from old to new

delete()
Delete all instructions in this block, so it can be removed

first_instruction
Return this blocks first instruction

insert_instruction(*instruction*, *before_instruction=None*)
Insert an instruction at the front of the block

is_closed
Determine whether this block is propert terminated

is_empty
Determines whether the block is empty or not

is_entry
Check if this block is the entry block of a function

is_used
True if this block is referenced by an instruction

last_instruction
Gets the last instruction from the block

phis
Return all *Phi* instructions of this block

predecessors
Return all predecessing blocks

remove_instruction(*instruction*)
Remove instruction from block

replace_incoming(*block*, *new_blocks*)
For each phi node in the block, change the incoming branch of block into new block with the same variable.

successors
Get the direct successors of this block

Types

class ppci.ir.**Typ**(*name*)
Built in type representation

is_blob
Test if this type is bytes blob

is_integer
Test if this type is of integer type

is_signed
Test if this type is of signed integer type

is_unsigned
Test if this type is of unsigned integer type

class ppci.ir.**BlobDataTyp**(*size: int*, *alignment: int*)
The type of a opaque data blob.

Note that blob types can be compared by using the is operator:

```
>>> from ppci.ir import BlobDataTyp
>>> typ1 = BlobDataTyp(8, 8)
>>> typ2 = BlobDataTyp(8, 8)
>>> typ1 is typ2
True
```

These simple types are available:

ppci.ir.**ptr**
Pointer type

ppci.ir.**i64**
Signed 64-bit type

ppci.ir.**i32**
Signed 32-bit type

ppci.ir.**i16**
Signed 16-bit type

ppci.ir.**i8**
Signed 8-bit type

ppci.ir.**u64**
Unsigned 64-bit type

ppci.ir.**u32**
Unsigned 32-bit type

ppci.ir.**u16**
Unsigned 16-bit type

ppci.ir.**u8**
Unsigned 8-bit type

ppci.ir.**f64**
64-bit floating point type

ppci.ir.**f32**
32-bit floating point type

Instructions

The following instructions are available.

Memory instructions

class ppci.ir.**Load**(*address*, *name*, *ty*, *volatile=False*)
Load a value from memory.

Parameters

- **address** – The address to load the value from.

- **name** – The name of the value after loading it from memory.
- **ty** – The type of the value.
- **volatile** – whether or not this memory access is volatile.

class ppci.ir.**Store**(*value*, *address*, *volatile=False*)
Store a value into memory

class ppci.ir.**Alloc**(*name: str*, *amount: int*, *alignment: int*)
Allocates space on the stack. The type of this value is a ptr

Data instructions

class ppci.ir.**Const**(*value*, *name*, *ty*)
Represents a constant value

class ppci.ir.**LiteralData**(*data*, *name*)
Instruction that contains labeled data. When generating code for this instruction, a label and its data is emitted in the literal area

class ppci.ir.**Binop**(*a*, *operation*, *b*, *name*, *ty*)
Generic binary operation

class ppci.ir.**Cast**(*value*, *name*, *ty*)
Base type conversion instruction

Control flow instructions

class ppci.ir.**ProcedureCall**(*callee*, *arguments*)
Call a procedure with some arguments

class ppci.ir.**FunctionCall**(*callee*, *arguments*, *name*, *ty*)
Call a function with some arguments and a return value

class ppci.ir.**Jump**(*target*)
Jump statement to another *Block* within the same function

class ppci.ir.**CJump**(*a*, *cond*, *b*, *lab_yes*, *lab_no*)
Conditional jump to true or false labels.

class ppci.ir.**Return**(*result*)
This instruction returns a value and exits the function.

This instruction is only legal in a *Function*.

class ppci.ir.**Exit**
Instruction that exits the procedure.

Note that this instruction can only be used in a *Procedure*.

Other

class ppci.ir.**Phi**(*name*, *ty*)
Imaginary phi instruction to make SSA possible.

The phi instruction takes a value input for each basic block which can reach the basic block in which this phi instruction is placed. So for each incoming branch, there is a value.

The phi instruction is an artificial instruction which allows the IR-code to be in SSA form.

class ppci.ir.**Undefined**(*name: str*, *ty: ppci.ir.Typ*)
Undefined value, this value must never be used.

Abstract instruction classes

There are some abstract instructions, which cannot be used directly but serve as base classes for other instructions.

class ppci.ir.**Instruction**
Base class for all instructions that go into a basic block

function
Return the function this instruction is part of

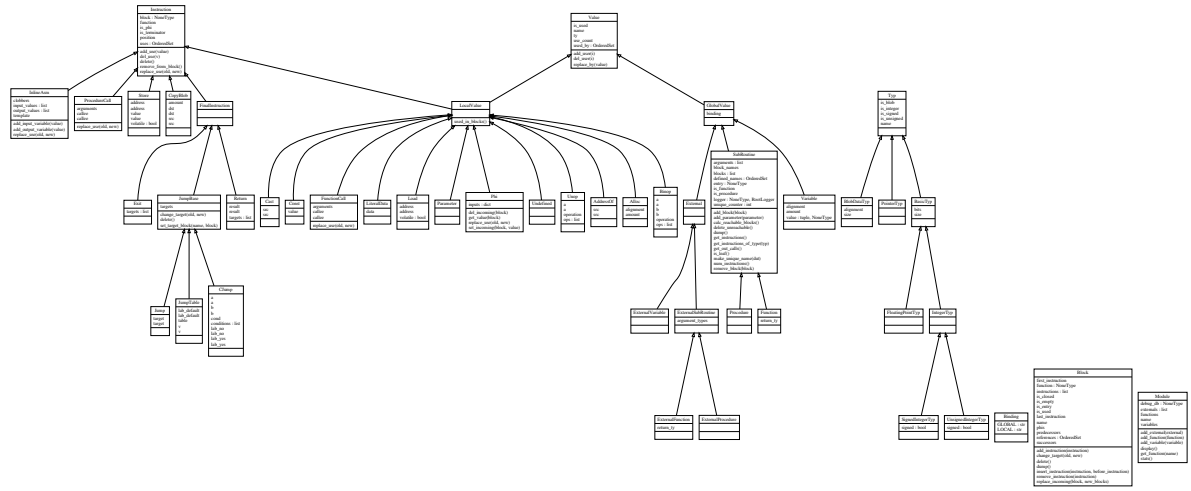
class ppci.ir.**Value**(*name: str*, *ty: ppci.ir.Typ*)
Base of all values

is_used
Determine whether this value is used anywhere

class ppci.ir.**FinalInstruction**
Final instruction in a basic block.

This instruction terminates the basic block. No more instruction may appear after this instruction.

Uml



3.7.2 Utilities

The `ppci.irutils` module contains function to handle IR-code such as `read_module()` and `verify_module()`. Also the `Builder` serves as a helper class to construct ir modules.

Module reference

Link two ir-modules, such that external references are resolved.

ppci.irutils.link.**ir_link**(*ir_modules*, *name='linked'*) → ppci.ir.Module
Link IR-modules into a single module.

Example:

```
>>> from ppci import ir
>>> from ppci.irutils import ir_link
>>> m1 = ir.Module('m1')
>>> m2 = ir.Module('m2')
>>> m3 = ir_link([m1, m2])
```

Note that the original modules are not usable after this action.

TODO: TBD: do not modify source modules?

Functions to add instrumentation to IR code.

ppci.irutils.instrument.**add_tracer**(*ir_module*, *trace_function_name='trace'*)
Instrument the given ir-module with a call tracer function

Constructing IR.

class ppci.irutils.builder.**Builder**
Helper class for IR-code generators.

This class can assist in the generation of IR-code. It's purpose is to simplify the language frontend, as well as to hide a bit of IR classes from the frontends.

emit(*instruction: ppci.ir.Instruction*) → ppci.ir.Instruction
Append an instruction to the current block

emit_add(*a*, *b*, *ty*)
Emit addition operation.

emit_binop(*a*, *op*, *b*, *ty*)
Emit a binary operation.

Parameters

- **a** – operand 1
- **op** – the operation to perform.
- **b** – operand 2, can be either a value or an int.
- **ty** – The type of a, b and the result.

Returns The result value of the binary operation.

emit_cast(*value*, *ty*)
Emit a type cast instruction.

emit_const(*value*, *ty*)
Emit a constant.

emit_exit()
Emit exit instruction.

emit_jump(*block*)
Emit a jump instruction to the given block.

emit_load(*address*, *ty*, *volatile=False*)
Emit a load instruction.

emit_mul(*a*, *b*, *ty*)
Emit multiplication operation.

emit_return(*value*)
Emit a return instruction.

emit_sub(*a*, *b*, *ty*)
Emit subtract operation.

new_block(*name=None*)
Create a new block and add it to the current function

new_function(*name*, *binding*, *return_ty*)
Create a new function.

new_procedure(*name*, *binding*)
Create a new procedure.

set_location(*location*)
Set the current source code location.

All instructions emitted from now on will be associated with the given sourcecode.

use_location(*location*)
Use the location for all code generated within a context.

ppci.irutils.builder.**split_block**(*block*, *pos=None*, *newname='splitblock'*)
Split a basic block into two which are connected.

Note to take care of phi instructions of successors and make sure to update those phi instructions.

3.7.3 JSON serialization

Module to serialize and deserialize IR-code.

Take an IR-module, and turn it into a dict or json. Then, this item can be persistet or send via e-mail. Next up, take this dict / json and reconstruct the identical IR-module from it.

This can be useful in these scenario's:

- Compilation caching: store the IR-code and load from disk when required later on.
- Distributed compilation: transfer IR-code across processes.

```
>>> import io
>>> from ppci.api import c_to_ir
>>> from ppci.irutils import to_json, from_json
>>> c_src = "int add(int a, int b) { return a + b; }"    # Define some C-code
>>> mod = c_to_ir(io.StringIO(c_src), "x86_64")          # turn C-code into IR-code
>>> mod.stats()
'functions: 1, blocks: 2, instructions: 11'
>>> json_txt = to_json(mod)                             # Turn module into JSON
>>> mod2 = from_json(json_txt)                          # Load module from JSON.
>>> mod2.stats()
'functions: 1, blocks: 2, instructions: 11'
```

class ppci.irutils.io.**DictReader**
Construct IR-module from given json.

get_value_ref(*name*, *ty=ir-typ ptr*)
Retrieve reference to a value.

class ppci.irutils.io.**DictWriter**
Serialize an IR-module as a dict.

ppci.irutils.io.**from_json**(*json_txt*)
Construct a module from valid json.

Parameters **json_txt** – A string with valid JSON.

Returns The IR-module as represented by JSON.

ppci.irutils.io.**to_json**(*module*)
Convert an IR-module to json format.

Parameters **module** – the IR-module intended for serialization.

Returns A JSON string representing the module.

3.7.4 Textual format

Parsing IR-code from a textual form.

exception ppci.irutils.reader.**IrParseException**
Exception raised when reading of IR-code fails.

class ppci.irutils.reader.**Reader**
Read IR-code from file

at_keyword(*keyword*)
Test if we are at some keyword.

consume_keyword(*keyword*)
Consume a specific identifier.

define_value(*value*)
Define a value

error(*msg*)
Something went wrong.

find_value(*name*, *ty=ir-typ i32*)
Try hard to find a value.

If the value is undefined, create a placeholder undefined value.

parse_assignment()
Parse an instruction with shape 'ty' 'name' '=' ...

parse_block(*function*)
Read a single block from file

parse_external(*module*)
Parse external variable.

parse_function(*binding*)
Parse a function or procedure

parse_module()
Entry for recursive descent parser

parse_statement()
Parse a single instruction line

parse_type()
Parse a single type

parse_value_ref(*ty=ir-typ ptr*)
Parse a reference to another variable.

read(*f*) → ppci.ir.Module
Read ir code from file f

ppci.irutils.reader.**read_module**(*f*) → ppci.ir.Module
Read an ir-module from file.

Parameters **f** – A file like object ready to be read in text modus.

Returns The loaded ir-module.

Writing IR-code into a textual format.

class ppci.irutils.writer.**Writer**(*file=None*, *extra_indent=''*)
Write ir-code to file

write(*module: ppci.ir.Module*, *verify=True*)
Write ir-code to file f

ppci.irutils.writer.**print_module**(*module*, *file=None*, *verify=True*)
Print an ir-module as text.

Parameters

- **module** (ir.Module) – The module to turn into textual format.
- **file** – An optional file like object to write to. Defaults to stdout.
- **verify** (*bool*) – A boolean indicating whether or not the module should be verified before writing.

3.7.5 Validation

Verify an IR-module for consistency.

This is a very useful module since it allows to isolate bugs in the compiler itself.

class ppci.irutils.verify.**Verifier**
Checks an ir module for correctness

block_dominates(*one: ppci.ir.Block*, *another: ppci.ir.Block*)
Check if this block dominates other block

instruction_dominates(*one*, *another*)
Checks if one instruction dominates another instruction

verify(*module*)
Verifies a module for some sanity

verify_block(*block*)
Verify block for correctness

verify_block_termination(*block*)
Verify that the block is terminated correctly

verify_function(*function*)
Verify all blocks in the function

verify_instruction(*instruction*, *block*)
Verify that instruction belongs to block and that all uses are preceeded by defs

verify_subroutine_call(*instruction*)
Check some properties of a function call

ppci.irutils.verify.**verify_module**(*module: ppci.ir.Module*)
Check if the module is properly constructed

Parameters **module** – The module to verify.

3.8 Optimization

The IR-code generated by the front-end can be optimized in many ways. The compiler does not have the best way to optimize code, but instead has a bag of tricks it can use.

3.8.1 Abstract base classes

The optimization passes all subclass one of the following base classes.

class ppci.opt.transform.**ModulePass**
Base class of all optimizing passes.

Subclass this class to implement your own optimization pass.

run(*ir_module*)
Run this pass over a module

class ppci.opt.transform.**FunctionPass**
Base pass that loops over all functions in a module

on_function(*function: ppci.ir.SubRoutine*)
Override this virtual method

run(*ir_module: ppci.ir.Module*)
Main entry point for the pass

class ppci.opt.transform.**BlockPass**
Base pass that loops over all blocks

on_block(*block: ppci.ir.Block*)
Override this virtual method

on_function(*function*)
Loops over each block in the function

class ppci.opt.transform.**InstructionPass**
Base pass that loops over all instructions

on_block(*block*)
Loops over each instruction in the block

on_instruction(*instruction*)
Override this virtual method

3.8.2 Optimization passes

class ppci.opt.**Mem2RegPromotor**
Tries to find alloc instructions only used by load and store instructions and replace them with values and phi nodes

class ppci.opt.**LoadAfterStorePass**
Remove load after store to the same location.

```
[x] = a
b = [x]
c = b + 2
```

transforms into:

```
[x] = a
c = a + 2
```

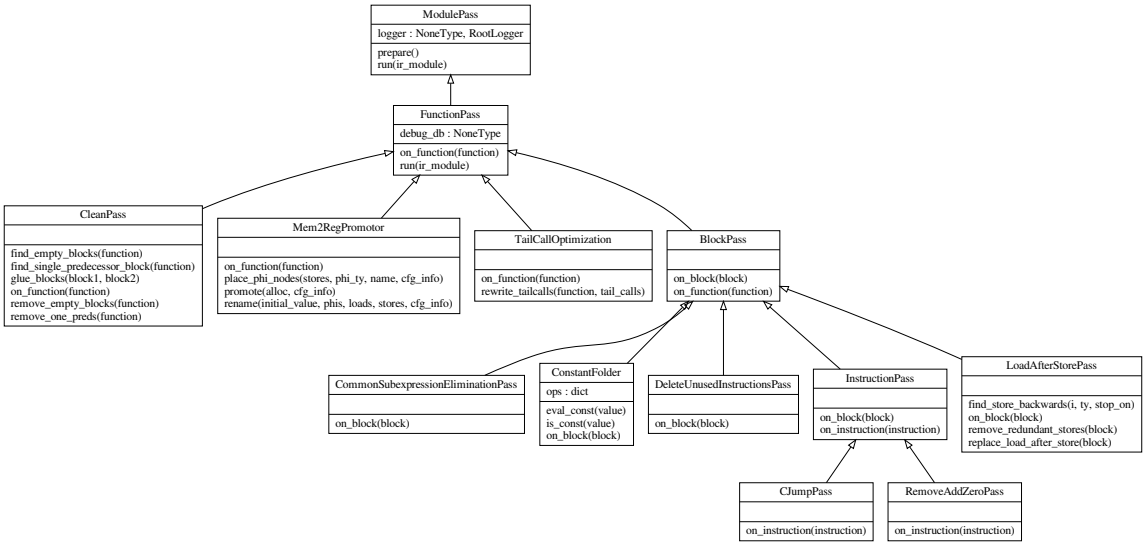
class ppci.opt.**DeleteUnusedInstructionsPass**
Remove unused variables from a block

class ppci.opt.**RemoveAddZeroPass**
Replace additions with zero with the value itself. Replace multiplication by 1 with value itself.

class ppci.opt.**CommonSubexpressionEliminationPass**
Replace common sub expressions (cse) with the previously defined one.

class ppci.opt.cjmp.**CJumpPass**

3.8.3 Uml



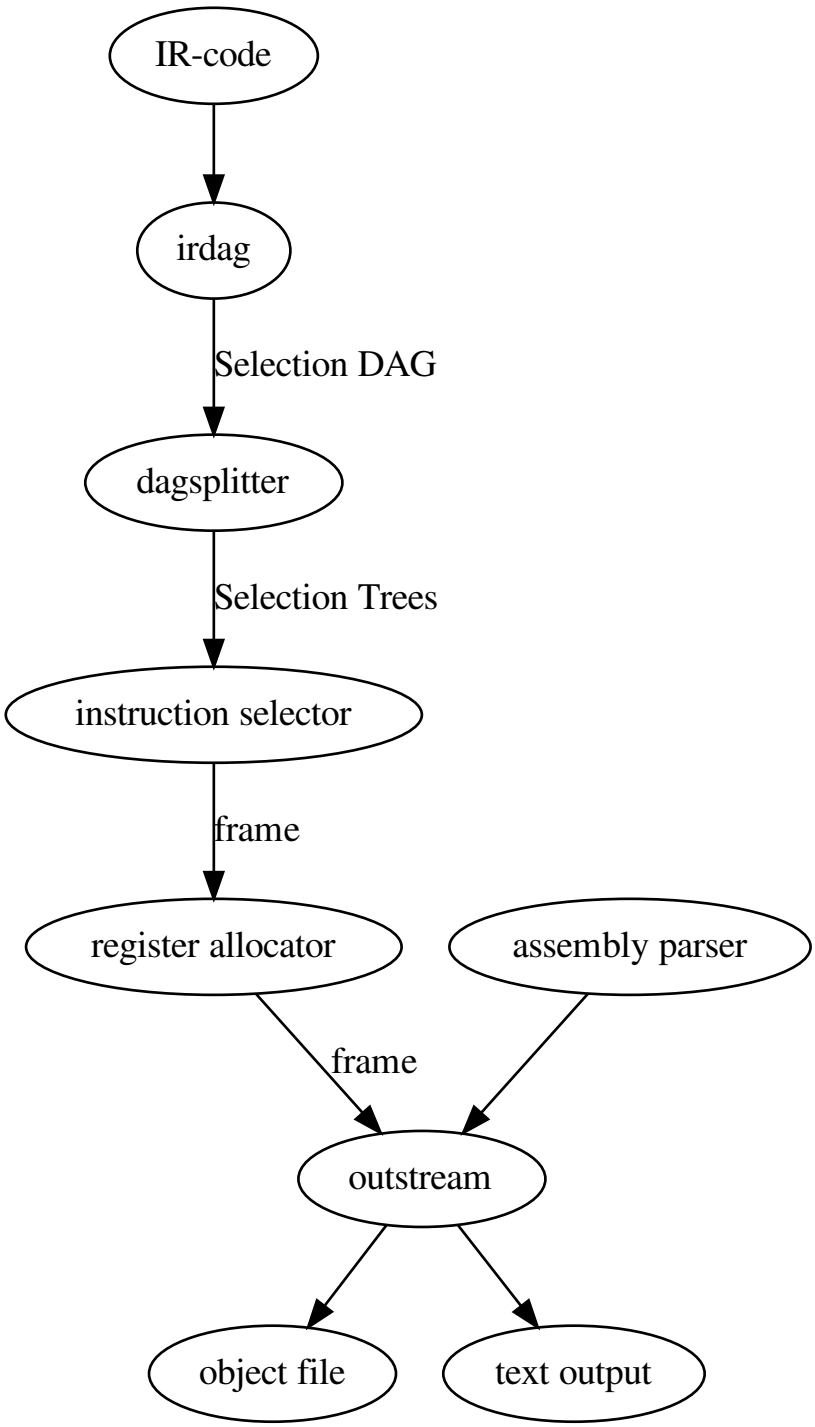
3.9 Code generation

The module *ppci.codegen* provides functions and classes to generate code.

3.9.1 Back-end

The back-end is more complicated. There are several steps to be taken here.

1. Tree creation
2. Instruction selection
3. Register allocation
4. Peep hole optimization



Canonicalize

During this phase, the IR-code is made simpler. Also unsupported operations are rewritten into function calls. For example, soft floating point is introduced here.

Code generator

Machine code generator.

The architecture is provided when the generator is created.

class ppci.codegen.codegen.**CodeGenerator**(*arch*, *reporter*, *optimize_for='size'*)
Machine code generator

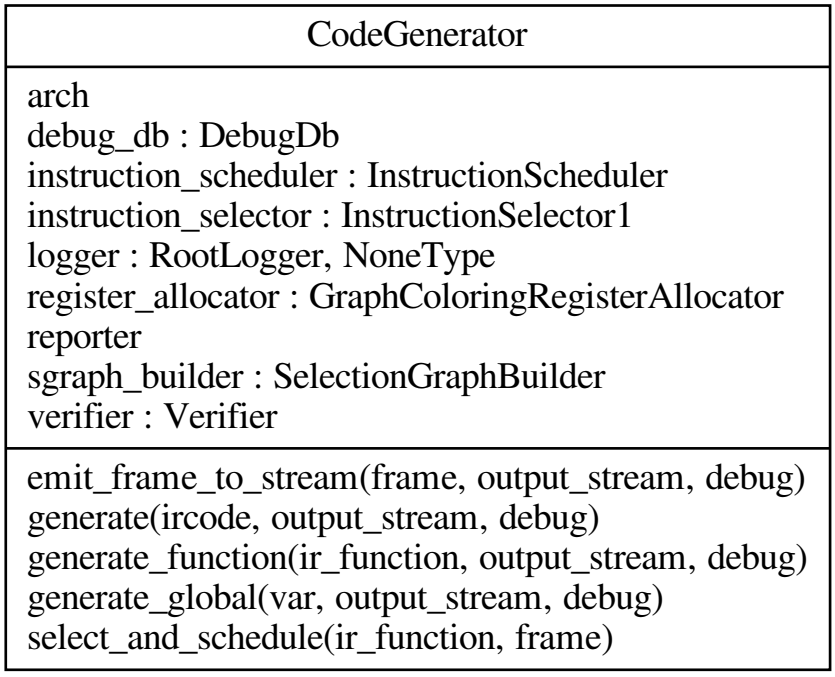
emit_frame_to_stream(*frame*, *output_stream*, *debug=False*)
Add code for the prologue and the epilogue. Add a label, the return instruction and the stack pointer adjustment for the frame. At this point we know how much stack space must be reserved for locals and what registers should be saved.

generate(*ircode: ppci.ir.Module*, *output_stream*, *debug=False*)
Generate machine code from ir-code into output stream

generate_function(*ir_function*, *output_stream*, *debug=False*)
Generate code for one function into a frame

generate_global(*var*, *output_stream*, *debug*)
Generate code for a global variable

select_and_schedule(*ir_function*, *frame*)
Perform instruction selection and scheduling



Instruction selection

The instruction selection phase takes care of scheduling and instruction selection. The output of this phase is a one frame per function with a flat list of abstract machine instructions.

To select instruction, a tree rewrite system is used. This is also called bottom up rewrite generator (BURG). See pyburg.

Instruction selector.

This part of the compiler takes in a DAG (directed acyclic graph) of instructions and selects the proper target instructions.

Selecting instructions from a DAG is a NP-complete problem. The simplest strategy is to split the DAG into a forest of trees and match these trees.

Another solution may be: PBQP (Partitioned Boolean Quadratic Programming)

The selection process creates operations in a selection DAG. The possible operations are listed in the below table:

operation	types	description
ADD(c0,c1)	I,U	Add its operands
SUB(c0,c1)	I,U	Substracts c1 from c0
MUL(c0,c1)	I,U	Multiplies c0 by c1
DIV(c0,c1)	I,U	Divides c0 by c1
OR(c0,c1)	I,U	Bitwise or
AND(c0,c1)	I,U	Bitwise and
XOR(c0,c1)	I,U	Bitwise exclusive or
LDR(c0)	I,U	Load from memory
STR(c0,c1)	I,U	Store value c1 at memory address c0
FPREL	U	Frame pointer relative location
CONST	I,U	Constant value
REG	I,U	Value in a specific register
JMP	I,U	Jump to a label
CJMP	I,U	Conditional jump to a label

...

Memory move operations:

- STRI64(REGI32[rax], CONSTI32[1])
- MOVB()

class ppci.codegen.instructionselector.**InstructionContext**(*frame*, *arch*)
Usable to patterns when emitting code

emit(*instruction*)
Abstract instruction emitter proxy

move(*dst*, *src*)
Generate move

new_label()
Generate a new unique label

new_reg(*cls*)
Generate a new temporary of a given class

class ppci.codegen.instructionselector.**InstructionSelector1**(*arch*, *sgraph_builder*, *reporter*, *weights=(1, 1, 1)*)
Instruction selector which takes in a DAG and puts instructions into a frame.

This one does selection and scheduling combined.

gen_tree(*context*, *tree*)
Generate code from a tree

inline_asm(*context*, *tree*)
Run assembler on inline assembly code.

memcp()
Invoke memcpy arch function

munch_trees(*context*, *trees*)
Consume a dag and match it using the matcher to the frame. DAG matching is NP-complete.

The simplest strategy is to split the dag into a forest of trees. Then, the DAG is reduced to only trees, which can be matched.

A different approach is use 0-1 programming, like the NOLTIS algo.

TODO: implement different strategies.

select(*ir_function: ppci.ir.SubRoutine*, *frame*)
Select instructions of function into a frame

class ppci.codegen.instructionselector.**TreeSelector**(*sys*)
Tree matcher that can match a tree and generate instructions

apply_rules(*context*, *tree*, *goal*)
Apply all selected instructions to the tree

burm_label(*tree*)
Label all nodes in the tree bottom up

gen(*context*, *tree*)
Generate code for a given tree. The tree will be tiled with patterns and the corresponding code will be emitted

kids(*tree*, *rule*)
Determine the kid trees for a rule

nts(*rule*)
Get the open ends of this rules pattern

Register allocation

Instructions generated during instruction selection phase use virtual registers and some physical registers (e.g. when an instruction expects arguments in particular register(s)). Register allocation is the process of assigning physical location (register or memory) to the remaining virtual registers.

Some key concepts in the domain of register allocation are:

- **virtual register**: A location which must be mapped to a physical register.
- **physical register**: A real CPU register
- **interference graph**: A graph in which each node represents a location. An edge indicates that the two locations cannot have the same physical register assigned.
- **pre-colored register**: A location that is already assigned a specific physical register.
- **coalescing**: The process of merging two nodes in an interference graph which do not interfere and are connected by a move instruction.
- **spilling**: Spilling is the process when no physical register can be found for a certain virtual register. Then this value will be placed in memory.
- **register class**: Most CPU's contain registers grouped into classes. For example, there may be registers for floating point, and registers for integer operations.
- **register alias**: Some registers may alias to registers in another class. A good example are the x86 registers rax, eax, ax, al and ah.

Interference graph

Each instruction in the instruction list may use or define certain registers. A register is live between a definition and a use of that register. Registers that are live at the same point in the program interfere with each other. An interference graph is a graph in which each node represents a register and each edge represents interference between those two registers.

Graph coloring

In 1981 Chaitin presented the idea to use graph coloring for register allocation.

In a graph a node can be colored if it has less neighbours than possible colors (referred to as K from now on). This is true because when each neighbour has a different color, there is still a valid color left for the node itself.

The outline of the algorithm is: Given a graph, if a node can be colored, remove this node from the graph and put it on a stack. When added back to the graph, it can be given a color. Now repeat this process recursively with the remaining graph. When the graph is empty, place all nodes back from the stack one by one and assign each node a color when placed. Remember that when adding back, a color can be found, because this was the criteria during removal.

See: https://en.wikipedia.org/wiki/Chaitin%27s_algorithm

[Chaitin1982]

Coalescing

Coalescing is the process of merging two nodes in an interference graph. This means that two temporaries will be assigned to the same register. This is especially useful if the temporaries are used in move instructions, since when the source and the destination of a move instruction are the same register, the move can be deleted.

Coalescing a move instruction is easy when an interference graph is present. Two nodes that are used by a move instruction can be coalesced when they do not interfere.

However, if we coalesce too many moves, the graph can become uncolorable, and spilling has to be done. To prevent spilling, coalescing must be done in a controlled manner.

A conservative approach to the coalescing is the following: if the merged node has fewer than K neighbours, then the nodes can be coalesced. The reason for this is that when all nodes that can be colored are removed and the merged node and its non-colored neighbours remain, the merged node can be colored. This ensures that the coalescing of the nodes does not have a negative effect on the colorability of the graph.

[Briggs1994]

Spilling

Iterated register coalescing

Iterated register coalescing (IRC) is a combination of graph coloring, coalescing and spilling. The process consists of the following steps:

- build an interference graph from the instruction list
- remove trivial colorable nodes.
- coalesce registers to remove redundant moves
- spill registers
- select registers

See: https://en.wikipedia.org/wiki/Register_allocation

[George1996]

Graph coloring with more register classes

Most instruction sets are not uniform, and hence simple graph coloring cannot be used. The algorithm can be modified to work with several register classes that possibly interfere.

[Runeson2003] [Smith2004]

Implementations

The following class can be used to perform register allocation.

class ppci.codegen.registerallocator.**GraphColoringRegisterAllocator**(*arch: ppci.arch.arch.Architecture*, *instruction_selector*, *reporter*)

Target independent register allocator.

Algorithm is iterated register coalescing by Appel and George. Also the pq-test algorithm for more register classes is added.

alloc_frame(*frame: ppci.arch.stack.Frame*)
Do iterated register allocation for a single frame.

This is the entry function for the register allocator and drives through all stages of the algorithm.

Parameters **frame** – The frame to perform register allocation on.

apply_colors()
Assign colors to registers

assign_colors()
Add nodes back to the graph to color it.

Any potential spills might turn into real spills at this stage.

calc_num_blocked(*node*)
Calculate for the given node how many registers are blocked by it's adjecent nodes.

This is an advanced form of a nodes degree, but then for register of different classes.

check_invariants()
Test invariants

coalesc()
Coalesc moves conservative.

This means, merge the variables of a move into one variable, and delete the move. But do this only when no spill will occur.

combine(*u*, *v*)
Combine u and v into one node, updating work lists

common_reg_class
Determine the smallest common register class of two nodes

conservative(*u*, *v*)
Briggs conservative criteria for coalescing.

If the result of the merge has fewer than K nodes that are not trivially colorable, then coalescing is safe, because when coloring, all other nodes that can be colored will be popped from the graph, leaving the merged node that then can be colored.

In the case of multiple register classes, first determine the new neighbour nodes. Then assume that all nodes that can be colored will be colored, and are taken out of the graph. Then calculate how many registers can be blocked by the remaining nodes. If this is less than the number of available registers, the coalesc is safe!

decrement_degree(*m*)
If a node was lowered in degree, check if there are nodes that can be moved from the spill list to the freeze of simplify list

freeze()
Give up coalescing on some node, move it to the simplify list and freeze all moves associated with it.

freeze_moves(*u*)
Freeze moves for node u

has_edge(*t*, *r*)
Helper function to check for an interfering edge

init_data(*frame: ppci.arch.stack.Frame*)
Initialize data structures

is_colorable(*node*) → bool
Helper function to determine whether a node is trivially colorable. This means: no matter the colors of the nodes neighbours, the node can be given a color.

In case of one register class, this is: n.degree < self.K

In case of more than one register class, somehow the worst case damage by all neighbours must be determined.

We do this now with the pq-test.

is_move_related(*n*)
Check if a node is used by move instructions

link_move(*move*)
Associate move with its source and destination

ok(*t*, *r*)
Implement coalescing testing with pre-colored register

q
The number of class B registers that can be blocked by class C.

release_pressure(*node*, *reg_class*)
Remove some register pressure from the given node.

remove_redundant_moves()
Remove coalesced moves

rewrite_program(*node*)
Rewrite program by creating a load and a store for each use

select_spill()
Select potential spill node.

Select a node to be spilled. This is optimistic, since this might be turned into a real spill. Continue nevertheless, to discover more potential spills, or we might be lucky and able to color the graph any ways.

simplify()
Remove nodes from the graph

class ppci.codegen.registerallocator.**MiniGen**(*arch*, *selector*)
Spill code generator

gen(*frame*, *tree*)
Generate code for a given tree

gen_load(*frame*, *vreg*, *slot*)
Generate instructions to load vreg from a stack slot

gen_store(*frame*, *vreg*, *slot*)
Generate instructions to store vreg at a stack slot

make_fmt(*vreg*)
Determine the type suffix, such as I32 or F64.

Peephole optimization

This page describes peephole optimization. The basic idea of peephole optimization is to take a small window (a peephole), and slide it over the instructions. Then, looking at this small sequence of say two or three instructions, check if this machine code sequence can be optimized.

For example, if we generated code in a previous pass which contains code like this:

```
jmp my_label
my_label:
```

It is easy to see that the jump instruction is not required. This kind of code can easily be generated by generating code for parts of the program and then concatenating them into the final code. The goal of the peephole optimizer is to detect these kind of constructions and in this case, remove the jump instruction, so that the final code will be like this:

```
my_label:
```

Another example is this code for the msp430 architecture:

These two instructions can be combined into one instruction which has the same effect:

Combiner

One way to implement a peephole optimizer is to write a lot of patterns and try all these patterns in turn. If a pattern matches a specific sequence of instructions, the pattern can be applied, and the instructions are substituted by the pattern substitute.

Another way, is to define per instruction the effects of the instruction, and for each pair of instructions that are evaulated, combine the effects of these instructions. If there exist an instruction which has the same effect as the combined effect of the two original instructions, the substitution can be made. This is the combiner approach as described by [Davidson1980].

The advantage of having the combiner, is that only per instructions the effects of the instruction must be defined. After this, all instructions with effects can be potentially combined. This reduces the amount of work to define peephole optimization patterns from N * N to N. Namely, not all instruction combinations must be described, but only the effects per instruction.

Module reference

Peephole optimization using the pipe-filter approach.

We face a certain stream of instructions. We take a look at a very specific window and check if we can apply the optimization. It's like scrolling over a sequence of instructions and checking for possible optimizations.

class ppci.codegen.peephole.**PeepHoleOptimization**
Inherit this class to implement a peephole optimization.

class ppci.codegen.peephole.**PeepHoleStream**(*downstream*)
This is a peephole optimizing output stream.

Having the peep hole optimizer as an output stream allows to use the peephole optimizer in several places.

clip_window(*size*)
Flush items, until we have *size* items in scope.

do_emit(*item*)
Actual emit implementation

flush()
Flush remaining items in the peephole window.

Code emission

Code is emitted using the outputstream class. The assembler and compiler use this class to emit instructions to. The stream can output to object file or to a logger.

class ppci.binutils.outstream.**OutputStream**
Interface to generate code with.

Contains the emit function to output instruction to the stream.

do_emit(*item*)
Actual emit implementation

emit(*item*)
Encode instruction and add symbol and relocation information

emit_all(*items*)
Emit all items from an iterable

select_section(*name*)
Switch output to certain section

Tree building

From IR-code a tree is generated which can be used to select instructions.

IR to DAG

The process of instruction selection is preceeded by the creation of a selection DAG (directed acyclic graph). The dagger take ir-code as input and produces such a dag for instruction selection.

A DAG represents the logic (computation) of a single basic block.

To do selection with tree matching, the DAG is then splitted into a series of tree patterns. This is often referred to as a forest of trees.

class ppci.codegen.irdag.**FunctionInfo**(*frame*)
Keeps track of global function data when generating code for part of a functions.

class ppci.codegen.irdag.**Operation**(*op*, *ty*)
A single operation with a type

class ppci.codegen.irdag.**SelectionGraphBuilder**(*arch*)
Create a selectiongraph from a function for instruction selection

block_to_sgraph(*ir_block: ppci.ir.Block*, *function_info*)
Create dag (directed acyclic graph) from a basic block.

The resulting dag can be used for instruction selection.

build(*ir_function: ppci.ir.SubRoutine*, *function_info*, *debug_db*)
Create a selection graph for the given function.

Selection graph is divided into groups for each basic block.

copy_phis_of_successors(*ir_block*)
When a terminator instruction is encountered, handle the copy of phi values into the expected virtual register

do_address_of(*node*)
Process ir.AddressOf instruction

do_alloc(*node*)
Process the alloc instruction

do_binop(*node*)
Visit a binary operator and create a DAG node

do_c_jump(*node*)
Process conditional jump into dag

do_cast(*node*)
Create a cast of type

do_const(*node*)
Process constant instruction

do_copy_blob(*node*)
Create a memcpy node.

do_function_call(*node*)
Transform a function call

do_inline_asm(*node*)
Create selection graph node for inline asm code.

This is a little weird, as we really do not need to select any instructions, but this special node will be filtered later on.

do_literal_data(*node*)
Literal data is stored after a label

do_load(*node*)
Create dag node for load operation

do_phi(*node*)
Refer to the correct copy of the phi node

do_procedure_call(*node*)
Transform a procedure call

do_return(*node*)
Move result into result register and jump to epilog

do_store(*node*)
Create a DAG node for the store operation

do_undefined(*node*)
Create node for undefined value.

do_unop(*node*)
Visit an unary operator and create a DAG node

get_address(*ir_address*)
Determine address for load or store.

new_node(*name*, *ty*, **args*, *value=None*)
Create a new selection graph node, and add it to the graph

new_vreg(*ty*)
Generate a new temporary fitting for the given type

ppci.codegen.irdag.**depth_first_order**(*function*)
Return blocks in depth first search order

ppci.codegen.irdag.**make_map**(*cls*)
Add an attribute to the class that is a map of ir types to handler functions. For example if a function is called do_phi it will be registered into f_map under key ir.Phi.

ppci.codegen.irdag.**prepare_function_info**(*arch*, *function_info*, *ir_function*)
Fill function info with labels for all basic blocks

DAG splitting into trees.

The selection dag is splitted into trees by this module.

class ppci.codegen.dagsplit.**DagSplitter**(*arch*)
Class that splits a DAG into a forest of trees.

This series is sorted such that data dependencies are met. The trees can henceforth be used to match trees.

assign_vregs(*sgraph*, *function_info*)
Give vreg values to values that cross block boundaries

check_vreg(*node*, *frame*)
Determine whether node outputs need a virtual register

get_reg_class(*data_flow*)
Determine the register class suited for this data flow line

make_op(*op*, *typ*)
Construct a string opcode from an operation and a type

make_trees(*nodes*, *tail_node*)
Create a tree from a list of sorted nodes.

split_into_trees(*sgraph*, *ir_function*, *function_info*, *debug_db*)
Split a forest of trees into a sorted series of trees for each block.

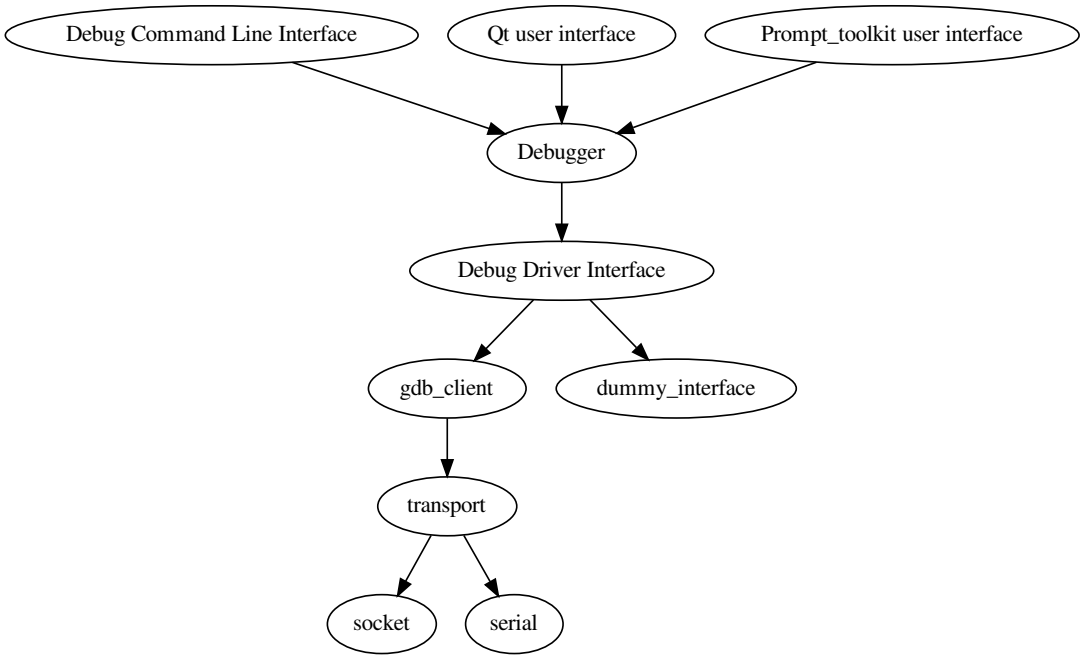
ppci.codegen.dagsplit.**topological_sort_modified**(*nodes*, *start*)
Modified topological sort, start at the end and work back

3.10 Debug

When an application is build, it often has to be debugged. This section describes the peculiarities of debugging.

3.10.1 Debugger

The debugger architecture is depicted below:



The debugger class is the main piece of the debugger. This is created for a specific architecture and is given a driver to communicate with the target hardware.

class ppci.binutils.dbg.**Debugger**(*arch*, *driver*)
Main interface to the debugger. Give it a target architecture for which it must debug and driver plugin to connect to hardware.

clear_breakpoint(*filename*, *row*)
Remove a breakpoint

read_mem(*address*, *size*)
Read binary data from memory location

run()
Run the program

set_breakpoint(*filename*, *row*)
Set a breakpoint

step()
Single step the debugged program

stop()
Interrupt the currently running program

write_mem(*address*, *data*)
Write binary data to memory location

One of the classes that uses the debugger is the debug command line interface.

class ppci.binutils.dbg.cli.**DebugCli**(*debugger*, *showsource=False*)
Implement a console-based debugger interface.

do_clrbrk(*arg*)
Clear a breakpoint. Specify the location by "filename, row" for example: main.c, 5

do_disasm(_)
Print disassembly around current location

do_info(_)
Show some info about the debugger

do_nstep(*count*)
Single instruction step the debugger

do_p(*arg*)
Print a variable

do_print(*arg*)
Print a variable

do_q(_)
Quit the debugger

do_quit(_)
Quit the debugger

do_read(*arg*)
Read data from memory: read address,length

do_readregs(_)
Read registers

do_restart(_)
Restart the running program

do_run(_)
Continue the debugger

do_s(_)
Single step the debugger

do_setbrk(*arg*)
Set a breakpoint: setbrk filename, row

do_setreg(*arg*)
Set registervalue

do_sl(*line*)
step one line

do_step(_)
Single step the debugger

do_stepi(_)
Single instruction step the debugger

do_stepl(*line*)
step one line

do_stop(_)
Stop the running program

do_write(*arg*)
Write data to memory: write address,hexdata

do_writeregs(_)
Write registers

postcmd(*stop*, *line*)
Hook method executed just after a command dispatch is finished.

precmd(*line*)
Hook method executed just before the command line is interpreted, but after the input prompt is generated and issued.

To connect to your favorite hardware, subclass the DebugDriver class.

class ppci.binutils.dbg.**DebugDriver**
Degug driver interface.

Inherit this class to expose a target interface. This class implements primitives for a given hardware target.

get_registers(*registers*)
Get the values for a range of registers

The following class can be used to connect to a gdb server:

class ppci.binutils.dbg.gdb.client.**GdbDebugDriver**(*arch*, *transport*, *pcresval=0*, *swbrkpt=False*)
Implement debugging via the GDB remote interface.

GDB servers can communicate via the RSP protocol.

Helpfull resources:

http://www.embecosm.com/appnotes/ean4/ embecosm-howto-rsp-server-ean4-issue-2.html

- https://sourceware.org/gdb/onlinedocs/gdb/Stop-Reply-Packets.html

Tried to make this class about the protocol itself, not about the sending and receiving of bytes. The protocol must be able to work using sockets and threads, serial port and threads and asyncio sockets.

clear_breakpoint(*address: int*)
Clear a breakpoint

connect()
Connect to the target

disconnect()
Disconnect the client

get_fp()
read the frame pointer

get_pc()
read the PC of the device

get_registers(*registers*)
Get the values for a range of registers

nstep(*count*)
Single step *count* times

read_mem(*address: int*, *size: int*)
Read memory from address

restart()
restart the device

run()
start the device

set_breakpoint(*address: int*)
Set a breakpoint

set_pc(*value*)
set the PC of the device

step()
Single step the device

write_mem(*address: int*, *data*)
Write memory

3.10.2 Debug info file formats

Debug information is of a complex nature. Various file formats exist to store this information. This section gives a short overview of the different formats.

pdb format

This is the microsoft debug format.

https://en.wikipedia.org/wiki/Program_database

Dwarf format

How a linked list is stored in dwarf format.

```
struct ll {
  int a;
  struct ll *next;
};
```

```
<1><57>: Abbrev Number: 3 (DW_TAG_base_type)
    <58>   DW_AT_byte_size   : 4
    <59>   DW_AT_encoding    : 5     (signed)
    <5a>   DW_AT_name        : int
 <1><5e>: Abbrev Number: 2 (DW_TAG_base_type)
    <5f>   DW_AT_byte_size   : 8
```

(continues on next page)

(continued from previous page)

```
   <60>   DW_AT_encoding    : 5    (signed)
   <61>   DW_AT_name        : (indirect string, offset: 0x65): long int
<1><65>: Abbrev Number: 2 (DW_TAG_base_type)
   <66>   DW_AT_byte_size   : 8
   <67>   DW_AT_encoding    : 7    (unsigned)
   <68>   DW_AT_name        : (indirect string, offset: 0xf6): sizetype
<1><6c>: Abbrev Number: 2 (DW_TAG_base_type)
   <6d>   DW_AT_byte_size   : 1
   <6e>   DW_AT_encoding    : 6    (signed char)
   <6f>   DW_AT_name        : (indirect string, offset: 0x109): char
<1><73>: Abbrev Number: 4 (DW_TAG_structure_type)
   <74>   DW_AT_name        : ll
   <77>   DW_AT_byte_size   : 16
   <78>   DW_AT_decl_file   : 1
   <79>   DW_AT_decl_line   : 4
   <7a>   DW_AT_sibling     : <0x95>
<2><7e>: Abbrev Number: 5 (DW_TAG_member)
   <7f>   DW_AT_name        : a
   <81>   DW_AT_decl_file   : 1
   <82>   DW_AT_decl_line   : 5
   <83>   DW_AT_type        : <0x57>
   <87>   DW_AT_data_member_location: 0
<2><88>: Abbrev Number: 6 (DW_TAG_member)
   <89>   DW_AT_name        : (indirect string, offset: 0xf1): next
   <8d>   DW_AT_decl_file   : 1
   <8e>   DW_AT_decl_line   : 6
   <8f>   DW_AT_type        : <0x95>
   <93>   DW_AT_data_member_location: 8
<2><94>: Abbrev Number: 0
<1><95>: Abbrev Number: 7 (DW_TAG_pointer_type)
   <96>   DW_AT_byte_size   : 8
   <97>   DW_AT_type        : <0x73>
```

3.11 Backends

This page lists the available backends.

3.11.1 Status

feature	mcs6500	arm	avr	msp430	or1k	riscv	stm8	x86_64	xtensa
processor architecture		von Neumann	modified Harvard	von Neumann				von Neumann	
native bitsize		32	8	16	32			64	
Assembly instructions		yes	yes	yes	yes	yes	yes	yes	yes
Samples build		yes	yes	yes	yes	yes		yes	yes
Samples run		yes		yes	yes			yes	yes
gdb remote client			yes			yes			
percentage complete	1%	70%	50%	20%	20%	70%	1%	60%	50%

3.11.2 Backend details

Processor architecture

The arch contains processor architecture descriptions.

class ppci.arch.arch.**Architecture**(*options=None*)
Base class for all targets

between_blocks(*frame*)
Generate any instructions here if needed between two blocks

determine_arg_locations(*arg_types*)
Determine argument location for a given function

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Generate instructions for the epilogue of a frame.

Parameters **frame** – the function frame for which to create a prologue

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Generate instructions for the epilogue of a frame.

Parameters **frame** – the function frame for which to create a prologue

get_compiler_rt_lib
Gets the runtime for the compiler. Returns an object with the compiler runtime for this architecture

get_reloc(*name*)
Retrieve a relocation identified by a name

get_reloc_type(*reloc_type*, *symbol*)
Re-implement this function to support ELF format relocations.

get_runtime()
Create an object with an optional runtime.

get_size(*typ*)
Get type of ir type

has_option(*name*)
Check for an option setting selected

make_id_str()
Return a string uniquely identifying this machine

move(*dst*, *src*)
Generate a move from src to dst

runtime
Gets the runtime for the compiler. Returns an object with the compiler runtime for this architecture

class ppci.arch.arch_info.**ArchInfo**(*type_infos=None*, *endianness=<Endianness.LITTLE: 1>*, *register_classes=()*)
A collection of information for language frontends

calc_alias()
Calculate a complete overview of register aliasing.

This uses the alias attribute when a register is defined.

For example on x86_64, *rax* aliases with *eax*, *eax* aliases *ax*, and *ax* aliases *al*.

This function creates a map from *al* to *rax* and vice versa.

get_alignment(*typ*)
Get the alignment for the given type

get_register(*name*)
Retrieve the machine register by name.

get_size(*typ*)
Get the size (in bytes) of the given type

get_type_info(*typ*)
Retrieve type information for the given type

has_register(*name*)
Test if this architecture has a register with the given name.

class ppci.arch.arch.**Frame**(*name*, *debug_db=None*, *fp_location=<FramePointerLocation.TOP: 1>*)
Activation record abstraction. This class contains a flattened function. Instructions are selected and scheduled at this stage. Frames differ per machine. The only thing left to do for a frame is register allocation.

add_constant(*value*)
Add constant literal to constant pool

add_out_call(*size*)
Record that we made a call out of this function.

The size parameter determines how much bytes we needed to reserve on the stack to pass the arguments.

alloc(*size: int*, *alignment: int*)
Allocate space on the stack frame and return a stacklocation

emit(*ins*)
Append an abstract instruction to the end of this frame

insert_code_after(*instruction*, *code*)
Insert a code sequence after an instruction

insert_code_before(*instruction*, *code*)
Insert a code sequence before an instruction

is_used(*register*, *alias*)
Check if a register or one of its aliases is used by this frame.

live_ranges(*vreg*)
Determine the live range of some register

new_label()
Generate a unique new label

new_name(*salt*)
Generate a new unique name

new_reg(*cls*, *twain=''*)
Retrieve a new virtual register

class ppci.arch.isa.**Isa**
Container type for an instruction set.

Contains a list of instructions, mappings from intermediate code to instructions.

Isa's can be merged into new isa's which can be used to define target. For example the arm without FPU can be combined with the FPU isa to expand the supported functions.

add_instruction(*instruction*)
Register an instruction into this ISA

pattern(*non_term*, *tree*, *condition=None*, *size=1*, *cycles=1*, *energy=1*)
Decorator function that adds a pattern.

peephole(*function*)
Add a peephole optimization function

register_pattern(*pattern*)
Add a pattern to this isa

register_relocation(*relocation*)
Register a relocation into this isa

class ppci.arch.registers.**Register**(*name*, *num=None*, *aliases=()*, *aka=()*)
Baseclass of all registers types

is_colored
Determine whether the register is colored

class ppci.arch.encoding.**Instruction**(**args*, ***kwargs*)
Base instruction class.

Instructions are created in the following ways:

- From python code, by using the instruction directly: self.stream.emit(Mov(r1, r2))
- By the assembler. This is done via a generated parser.
- By the instruction selector. This is done via pattern matching rules

Instructions can then be emitted to output streams.

Instruction classes are automatically added to an isa if they have an isa attribute.

classmethod decode(*data*)
Decode data into an instruction of this class

defined_registers
Return a set of all defined registers

encode()
Encode the instruction into binary form.

returns bytes for this instruction.

get_positions()
Calculate the positions in the byte stream of all parts

reads_register(*register*)
Check if this instruction reads the given register

registers
Determine all registers used by this instruction

relocations()
Determine the total set of relocations for this instruction

replace_register(*old*, *new*)
Replace a register usage with another register

set_all_patterns(*tokens*)
Look for all patterns and apply them to the tokens

classmethod sizes()
Get possible encoding sizes in bytes

used_registers
Return a set of all registers used by this instruction

writes_register(*register*)
Check if this instruction writes the given register

arm

Arm machine specifics. The arm target has several options:

- thumb: enable thumb mode, emits thumb code

class ppci.arch.arm.**ArmArch**(*options=None*)
Arm machine class.

between_blocks(*frame*)
Generate any instructions here if needed between two blocks

determine_arg_locations(*arg_types*)
Given a set of argument types, determine location for argument ABI: pass arg1 in R1 pass arg2 in R2 pass arg3 in R3 pass arg4 in R4 return value in R0

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Return epilogue sequence for a frame.

Adjust frame pointer and add constant pool.

Also free up space on stack for:

- Space for parameters passed to called functions.
- Space for save registers
- Space for local variables

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Returns prologue instruction sequence.

Reserve stack for this calling frame for:

- local variables
- save registers
- parameters to called functions

get_runtime()
Implement compiler runtime functions

litpool(*frame*)
Generate instruction for the current literals

move(*dst*, *src*)
Generate a move from src to dst

avr

Testing

To test the avr code, the simavr project is used.

https://github.com/buserror/simavr

Module

The is the avr backend.

See also:

https://gcc.gnu.org/wiki/avr-gcc

http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_instruction_list.html

class ppci.arch.avr.**AvrArch**(*options=None*)
Avr architecture description.

between_blocks(*frame*)
Generate any instructions here if needed between two blocks

determine_arg_locations(*arg_types*)
Given a set of argument types, determine location for argument

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Return epilogue sequence for a frame. Adjust frame pointer and add constant pool

gen_function_enter(*args*)
Copy arguments into local temporaries and mark registers live

gen_prologue(*frame*)
Generate the prologue instruction sequence.

get_runtime()
Create an object with an optional runtime.

litpool(*frame*)
Generate instruction for the current literals

move(*dst*, *src*)
Generate a move from src to dst

class ppci.arch.avr.**AvrRegister**(*name*, *num=None*, *aliases=()*, *aka=()*)
An 8-bit avr register

classmethod from_num(*num*)
Retrieve the singleton instance of the given register number.

class ppci.arch.avr.**AvrWordRegister**(*name*, *num=None*, *aliases=()*, *aka=()*)
Register covering two 8 bit registers

classmethod from_num(*num*)
Retrieve the singleton instance of the given register number.

m68k

This architecture is also known as M68000.

M68000 architecture.

class ppci.arch.m68k.**M68kArch**(*options=None*)

determine_arg_locations(*arg_types*)
Determine argument locations.

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Return epilogue sequence for a frame.

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Generate the prologue instruction sequence

move(*dst*, *src*)
Generate a move from src to dst

Microblaze

The microblaze is a softcore architecture specified by xilinx. It is a 32 bit processor with 32 registers.

See also:

https://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

Reference

Microblaze architecture.

Low level instruction class usage:

```
>>> from ppci.arch.microblaze import instructions, registers
>>> i = instructions.Add(registers.R4, registers.R5, registers.R6)
>>> str(i)
'add R4, R5, R6'
```

class ppci.arch.microblaze.**MicroBlazeArch**(*options=None*)
Microblaze architecture

determine_arg_locations(*arg_types*)
Use registers R5-R10 to pass arguments

determine_rv_location(*ret_type*)
Return values in R3-R4

gen_call(*frame*, *label*, *args*, *rv*)
Generate proper calling sequence

gen_epilogue(*frame*)
Return epilogue sequence for a frame.

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_litpool(*frame*)
Generate instructions for literals

gen_prologue(*frame*)
Returns prologue instruction sequence.

get_runtime()
Compiles the runtime support for microblaze.

It takes some c3 code.

move(*dst*, *src*)
Generate a move from src to dst

class ppci.arch.microblaze.**MicroBlazeRegister**(*name*, *num=None*, *aliases=()*, *aka=()*)
A microblaze register

classmethod from_num(*num*)
Retrieve the singleton instance of the given register number.

mcs6500

The famous 6502!

class ppci.arch.mcs6500.**Mcs6500Arch**(*options=None*)
6502 architectur

determine_arg_locations(*arg_types*)

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Generate instructions for the epilogue of a frame.

Parameters **frame** – the function frame for which to create a prologue

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Generate instructions for the epilogue of a frame.

Parameters **frame** – the function frame for which to create a prologue

msp430

Testing

To test the code for msp430, the openmsp430 project is used.

https://opencores.org/project,openmsp430

Module

To flash the msp430 board, the following program can be used:

http://www.ti.com/tool/msp430-flasher

class ppci.arch.msp430.**Msp430Arch**(*options=None*)
Texas Instruments msp430 target architecture

determine_arg_locations(*arg_types*)
Given a set of argument types, determine location for argument ABI: param1 = r12 param2 = r13 param3 = r14 param4 = r15 further parameters are put on stack. retval = r12

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Return epilogue sequence for a frame. Adjust frame pointer and add constant pool

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Returns prologue instruction sequence

get_runtime()
Compiles the runtime support for msp430. It takes some c3 code and some assembly helpers.

litpool(*frame*)
Generate instruction for the current literals

move(*dst*, *src*)
Generate a move from src to dst

static round_upwards(*value*)
Round value upwards to multiple of 2

MIPS

Module

Mips architecture

class ppci.arch.mips.**MipsArch**(*options=None*)
Mips architecture

determine_arg_locations(*arg_types*)
Determine argument location for a given function

determine_rv_location(*ret_type*)
return value in v0-v1

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Return epilogue sequence

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Returns prologue instruction sequence

get_runtime()
Retrieve the runtime for this target

move(*dst*, *src*)
Generate a move from src to dst

Open risc

Qemu

When booting with qemu, the loading of a raw binary goes wrong as of qemu version 2.11. Instead of loading a raw binary, an uboot image can be created with `ppci.utils.uboot_image.write_uboot_image()`.

```
$ qemu-system-or1k -kernel baremetal.uimage -M or1k-sim -serial stdio -m 16M
```

The memory is mapped as follows:

```
(qemu) info mtree
address-space: memory
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000000000-0000000000ffffff (prio 0, ram): openrisc.ram
    0000000090000000-0000000090000007 (prio 0, i/o): serial
    0000000092000000-0000000092000053 (prio 0, i/o): open_eth.regs
    0000000092000400-00000000920007ff (prio 0, i/o): open_eth.desc

address-space: I/O
  0000000000000000-000000000000ffff (prio 0, i/o): io

address-space: cpu-memory
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000000000-0000000000ffffff (prio 0, ram): openrisc.ram
    0000000090000000-0000000090000007 (prio 0, i/o): serial
    0000000092000000-0000000092000053 (prio 0, i/o): open_eth.regs
    0000000092000400-00000000920007ff (prio 0, i/o): open_eth.desc
```

To get a lot of debug output, the trace option of qemu can be used:

```
-D trace.txt -d in_asm,exec,int,op_opt,cpu
```

Module

Open risk architecture target.

class ppci.arch.or1k.**Or1kArch**(*options=None*)
Open risc architecture.

ABI: r0 -> zero r1 -> stack pointer r2 -> frame pointer r3 -> parameter 0 r4 -> parameter 1 r5 -> parameter 2 r6 -> parameter 3 r7 -> parameter 4 r8 -> parameter 6 r9 -> link address (return address for functions) r11 -> return value

determine_arg_locations(*arg_types*)
Given a set of argument types, determine location for argument

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Return epilogue sequence for a frame.

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Generate the prologue instruction sequence

litpool(*frame*)
Generate instructions for literals

move(*dst*, *src*)
Generate a move from src to dst

risc-v

Backend for the risc-v platform. See also: https://riscv.org/

Testing

To test the riscv platform, the picorv32 project is used.

https://github.com/cliffordwolf/picorv32

Module

See also: http://riscv.org

Contributed by Michael.

class ppci.arch.riscv.**RiscvArch**(*options=None*)

between_blocks(*frame*)
Generate any instructions here if needed between two blocks

determine_arg_locations(*arg_types*)
Given a set of argument types, determine location for argument ABI: pass args in R12-R17 return values in R10

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*frame*, *label*, *args*, *rv*)
Implement actual call and save / restore live registers

gen_epilogue(*frame*)
Return epilogue sequence for a frame. Adjust frame pointer and add constant pool

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Returns prologue instruction sequence

get_runtime()
Implement compiler runtime functions

litpool(*frame*)
Generate instruction for the current literals

move(*dst*, *src*)
Generate a move from src to dst

stm8

Stm8 support.

STM8 is an 8-bit processor, see also:

http://www.st.com/stm8

Implementation

Since there are so few registers, one possible approach is to emulate registers in the first 16 or 32 bytes of ram. We can then use these ram locations as 'registers' and do register allocation and instruction selection using these 'registers'. This way, we treat the stm8 as a risc machine with many registers, while in reality it is not.

Calling conventions

There is not really a standard calling convention for the stm8 processor.

Since the stm8 has only few registers, a calling convention must place most arguments on stack.

class ppci.arch.stm8.**Stm8Arch**(*options=None*)
STM8 architecture description.

determine_arg_locations(*arg_types*)
Calling convention in priority order:

- Pointers in index registers;
- 16-bit variables in index registers;
- 8-bit variables in accumulator register first, afterwards in index registers.

determine_rv_location(*ret_type*)
Determine the location of a return value of a function given the type of return value

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Generate instructions for the epilogue of a frame.

Parameters **frame** – the function frame for which to create a prologue

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Generate instructions for the epilogue of a frame.

Parameters **frame** – the function frame for which to create a prologue

x86_64

For a good list of op codes, checkout:

http://ref.x86asm.net/coder64.html

For an online assembler, checkout:

https://defuse.ca/online-x86-assembler.htm

Linux

For a good list of linux system calls, refer:

http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64

class ppci.arch.x86_64.**X86_64Arch**(*options=None*)
x86_64 architecture

determine_arg_locations(*arg_types*)

Given a set of argument types, determine locations the first arguments go into registers. The others on the stack.

see also http://www.x86-64.org/documentation/abi.pdf

ABI: p1 = rdi p2 = rsi p3 = rdx p4 = rcx p5 = r8 p6 = r9

floating point values are passed in xmm0, xmm1, xmm2, xmm3, etc..

return value in rax

self.rv = rax

On windows a different scheme is used: integers are passed in rcx, rdx, r8 and r9 floats are passed in xmm0, xmm1, xmm2 and xmm3

These examples show how it works:

func(int a, double b, int c, float d) // a in rcx, b in xmm1, c in r8 and d in xmm3

determine_rv_location(*ret_type*)
return value in rax or xmm0

gen_call(*frame*, *label*, *args*, *rv*)
This function moves arguments in the proper locations.

gen_epilogue(*frame*)
Return epilogue sequence for a frame. Adjust frame pointer and add constant pool

gen_function_enter(*args*)
Copy arguments into local temporaries and mark registers live

gen_memcpy(*dst*, *src*, *count*)
Generate a memcpy action

gen_prologue(*frame*)
Returns prologue instruction sequence

get_reloc_type(*reloc_type*, *symbol*)
Get the reloc type for ELF format.

move(*dst*, *src*)
Generate a move from src to dst

xtensa

Module

The xtensa architecture

Xtensa is used in the esp8266 and esp32 chips from espressif.

class ppci.arch.xtensa.**XtensaArch**(*options=None*)
Xtensa architecture implementation.

determine_arg_locations(*arg_types*)
Determine argument location for a given function

determine_rv_location(*ret_type*)
return value in a2

gen_call(*frame*, *label*, *args*, *rv*)
Generate instructions for a function call.

gen_epilogue(*frame*)
Return epilogue sequence

gen_function_enter(*args*)
Generate code to extract arguments from the proper locations

The default implementation tries to use registers and move instructions.

Parameters **args** – an iterable of virtual registers in which the arguments must be placed.

gen_prologue(*frame*)
Returns prologue instruction sequence

get_runtime()
Retrieve the runtime for this target

move(*dst*, *src*)
Generate a move from src to dst

Testing

The xtensa backend

You can run xtensa with qemu:

```
$ qemu-system-xtensa -M lx60 -m 96M -pflash lx60.flash -serial stdio
```

This will run the lx60 emulated board. This is an Avnet board with an fpga and an emulated xtensa core on it. This board can boot from parallel flash (pflash).

The memory is mapped as follows, you can see it in the qemu monitor with the 'info mtree' command:

```
(qemu) info mtree
address-space: memory
  0000000000000000-ffffffffffffffff (prio 0, RW): system
    0000000000000000-0000000005ffffff (prio 0, RW): lx60.dram
    00000000f0000000-00000000fdffffff (prio 0, RW): lx60.io
      00000000f8000000-00000000f83fffff (prio 0, R-): lx60.io.flash
      00000000fd020000-00000000fd02ffff (prio 0, RW): lx60.fpga
      00000000fd030000-00000000fd030053 (prio 0, RW): open_eth.regs
      00000000fd030400-00000000fd0307ff (prio 0, RW): open_eth.desc
      00000000fd050020-00000000fd05003f (prio 0, RW): serial
      00000000fd800000-00000000fd803fff (prio 0, RW): open_eth.ram
    00000000fe000000-00000000fe3fffff (prio 0, RW): alias lx60.flash @lx60.io.
↪flash 0000000000000000-00000000003fffff

address-space: I/O
  0000000000000000-000000000000ffff (prio 0, RW): io

address-space: cpu-memory
  0000000000000000-ffffffffffffffff (prio 0, RW): system
    0000000000000000-0000000005ffffff (prio 0, RW): lx60.dram
    00000000f0000000-00000000fdffffff (prio 0, RW): lx60.io
      00000000f8000000-00000000f83fffff (prio 0, R-): lx60.io.flash
      00000000fd020000-00000000fd02ffff (prio 0, RW): lx60.fpga
      00000000fd030000-00000000fd030053 (prio 0, RW): open_eth.regs
      00000000fd030400-00000000fd0307ff (prio 0, RW): open_eth.desc
      00000000fd050020-00000000fd05003f (prio 0, RW): serial
      00000000fd800000-00000000fd803fff (prio 0, RW): open_eth.ram
    00000000fe000000-00000000fe3fffff (prio 0, RW): alias lx60.flash @lx60.io.
↪flash 0000000000000000-00000000003fffff

memory-region: lx60.io.flash
  0000000008000000-00000000083fffff (prio 0, R-): lx60.io.flash
```

The lx60 emulation has also an mmu, which means it uses a memory translation unit. Therefore, the RAM is mapped from 0xd8000000 to 0x00000000.

A working memory map for this target is:

```
MEMORY flash LOCATION=0xfe000000 SIZE=0x10000 {
    SECTION(reset)
    ALIGN(4)
    SECTION(code)
}

MEMORY ram LOCATION=0xd8000000 SIZE=0x10000 {
    SECTION(data)
}
```

Code starts to run from the first byte of the flash image, and is mapped at 0xfe000000.

3.12 File formats

3.12.1 Elf

The executable and link format used on many linux systems.

Reference

ELF file format module

ppci.format.elf.**read_elf**(*f*)
Read an ELF file

ppci.format.elf.**write_elf**(*obj*, *f*, *type='executable'*)
Save object as an ELF file.

You can specify the type of ELF file with the type argument: - 'executable' - 'relocatable'

class ppci.format.elf.**ElfFile**(*bits=64*, *endianness=<Endianness.LITTLE: 1>*)
This class can load and save a elf file.

get_str(*offset*)
Get a string indicated by numeric value

3.12.2 Exe files

The following module can help when working with exe files.

Implements the exe file format used by windows.

See also:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx

Another excellent effort: https://github.com/erocarrera/pefile

ppci.format.exefile.**align**(*f*, *alignment*, *padding_value=0*)
Pad with 0 bytes until certain alignment is met

ppci.format.exefile.**read_exe**(*f*)
Process an exe file

ppci.format.exefile.**write_dos_stub**(*f*)
Write dos stub to file. This stub prints to stdout: 'this program cannot be run in dos mode'

3.12.3 Dwarf

Dwarf is a widely used debugging format.

Module reference

Implementation of the dwarf debugging format.

See also: http://dwarfstd.org/

3.12.4 Hexfile manipulation

```
>>> from ppci.format.hexfile import HexFile
>>> h = HexFile()
>>> h.add_region(0, bytes([1, 2, 3]))
>>> h.add_region(0x100, 'Hello world of hex'.encode('utf8'))
>>> h
Hexfile containing 21 bytes
>>> h.dump()
Hexfile containing 21 bytes
Region at 0x00000000 of 3 bytes
00000000  01 02 03                                          |...|
Region at 0x00000100 of 18 bytes
00000100  48 65 6c 6c 6f 20 77 6f  72 6c 64 20 6f 66 20 68  |Hello.world.of.h|
00000110  65 78                                             |ex|
```

Reference

Module to work with intel hex files.

This module can be used to work with intel hexfiles.

class ppci.format.hexfile.**HexFile**
Represents an intel hexfile

add_region(*address*, *data*)
Add a chunk of data at the given address

dump(*contents=False*)
Print info about this hexfile

static load(*open_file*)
Load a hexfile from file

save(*f*)
Save hexfile to file-like object

write_hex_line(*line*)
Write a single hexfile line

exception ppci.format.hexfile.**HexFileException**
Exception raised when hexfile handling fails

class ppci.format.hexfile.**HexFileRegion**(*address*, *data=b''*)
A continuous region of data starting at some address

add_data(*data*)
Add data to this region

end_address
End address for this region

size
The size of this region

class ppci.format.hexfile.**HexLine**(*address*, *typ*, *data=b''*)
A single line in a hexfile

classmethod from_line(*line: str*)
Parses a hexfile line into three parts

to_line() → str
Create an ascii hex line

3.12.5 Hunk amiga files

Amiga hunk format.

See also:

https://github.com/cnvogelg/amitools/tree/master/amitools/binfmt/hunk

ppci.format.hunk.**read_hunk**(*filename*)
Read a hunk file.

ppci.format.hunk.**write_hunk**(*filename*, *code_data*)
Write a hunk file.

3.12.6 uboot image files

Uboot is a popular bootloader for embedded linux devices. This module can be used to create uboot files.

Module reference

Uboot image file format

class ppci.format.uboot_image.**ApplicationType**
Application type

class ppci.format.uboot_image.**Architecture**
Computer architecture

class ppci.format.uboot_image.**Compression**
Compression types

class ppci.format.uboot_image.**ImageHeader**

class ppci.format.uboot_image.**OperatingSystem**
Operating system

ppci.format.uboot_image.**write_uboot_image**(*f*, *data: bytes*, *image_name='foobar'*, *load_address=256*, *entry_point=256*, *os=<OperatingSystem.INVALID: 0>*, *arch=<Architecture.OPENRISC: 21>*)
Write uboot image to file

3.12.7 S-record

Motorola s-record format.

https://en.wikipedia.org/wiki/SREC_(file_format)

ppci.format.srecord.**write_srecord**(*obj*, *f*)
Write object to srecord

3.13 Web Assembly

Web Assembly (wasm) is a portable binary format designed for the web.

http://webassembly.org/

3.13.1 Creating a wasm module

A WASM *Module* can be created from its text representation, a corresponding tuple structure, a bytes object or binary file, or from another Module object:

```
>>> from ppci import wasm
>>> code = '(module (func $truth (result i32) (i32.const 42) (return)))'
>>> m1 = wasm.Module(code)
>>> m2 = wasm.Module(m1.to_bytes())
```

The *read_wasm()* function is a thin wrapper around the Module constructor.

3.13.2 Exporting a wasm module

A wasm module can be exported to text or binary form:

```
>>> code = '(module (func $truth (result i32) (i32.const 42) (return)))'
>>> m = wasm.Module(code)
>>> m.to_string()
'(module\n  (type $0 (func (result i32)))\n  (func $truth (type $0)\n    i32.const␣
↪42\n    return)\n)\n'
```

(continues on next page)

(continued from previous page)

```
>>> m.to_bytes()
b
↪'\x00asm\x01\x00\x00\x00\x01\x05\x01`\x00\x01\x7f\x03\x02\x01\x00\n\x07\x01\x05\x00A*\x0f\x0b
↪'
```

And can also be "displayed" in these forms:

```
>>> m.show()
(module
  (type $0 (func (result i32)))
  (func $truth (type $0)
    i32.const 42
    return)
)

>>> m.show_bytes()
00000000  00 61 73 6d 01 00 00 00  01 05 01 60 00 01 7f 03  |.asm.......`....|
00000010  02 01 00 0a 07 01 05 00  41 2a 0f 0b              |........A*..|
```

3.13.3 Running wasm

Wasm can be executed in node:

```
wasm.run_wasm_in_node(m)
```

Or in the browser by exporing an html file:

```
wasm.export_wasm_example('~/wasm.html', code, m)
```

Inside a jupyter notebook, the WASM can be run directly:

```
wasm.run_wasm_in_notebook(m)
```

Running in the Python process:

```
>>> code = '(module (func (export truth) (result i32) (i32.const 42) (return)))'
>>> m1 = wasm.Module(code)
>>> imports = {}  # Python function imports can be added here
>>> loaded = wasm.instantiate(m1, imports)
>>> loaded.exports.truth()
42
```

Running WASI modules can be done *from command line*:

```
$ python -m ppci.cli.wabt run --target native coremark-wasi.wasm
```

Note that you can pass arguments to the WASI executable by using double dash:

    $ python -m ppci.cli.wabt run –target native coremark-wasi.wasm – -h

3.13.4 Converting between wasm and ir

With the *ppci.wasm.wasm_to_ir()* class it is possible to translate wasm code to ir-code. It is also possible to translate ir-code into wasm with the *ppci.wasm.ir_to_wasm()* function. This allows, for instance, running C3 on the web, or Web Assembly on a microprocessor.

The basic functionality is there, but more work is needed e.g. a stdlib for this functionality to be generally useful.

```
>>> from ppci import wasm
>>> from ppci.wasm.arch import WasmArchitecture
>>> code = '(module (func $truth (result i32) (i32.const 42) (return)))'
>>> m1 = wasm.Module(code)
>>> arch = WasmArchitecture()
>>> ir = wasm.wasm_to_ir(m1, arch.info.get_type_info('ptr'))
>>> m2 = wasm.ir_to_wasm(ir)
```

3.13.5 Module reference

Tools for representing, loading and exporting WASM (Web Assembly), and for converting between PPCI-IR and WASM.

class ppci.wasm.**WASMComponent**(**input*)

Base class for representing components of a WASM module, from the Module to Imports, Funct and Instruction. These components can be shown as text or written as bytes.

Each component can be instantiated using:

- its attributes (the most direct method).
- a tuple representing an S-expression.
- a string representing an S-expression.
- a bytes object representing the binary form of a component.
- a file object that contains the binary form of a component.

show()

Print the S-expression of the component.

to_string()

Get the textual representation (S-expression) of this component.

to_tuple()

Get the component's tuple representation (by exporting to string and parsing the s-expression).

class ppci.wasm.**Instruction**(**input*)

Class ro represent an instruction (an opcode plus arguments).

to_string()

Get the textual representation (S-expression) of this component.

class ppci.wasm.**BlockInstruction**(**input*)

An instruction that represents a block of instructions. (block, loop or if). The args consists of a single element indicating the result type. It can optionally have an id.

to_string()

Get the textual representation (S-expression) of this component.

class ppci.wasm.**Module**(**input*)

Class to represent a WASM module; the toplevel unit of code.

The Module is a collection of definitions, which can be provided as text, tuples, or Definition objects.

add_definition(*d*)

Add a definition to the module.

get_definitions_per_section()

Get a dictionary that maps section names to definitions. Note that the 'code' section is named 'func'. Note that one can also use e.g. `module['type']` to get all typedefs.

show_bytes()

Show the binary representation of this WASM module.

show_interface()
Show the (signature of) imports and exports in a human friendly manner.

to_bytes()
Get the bytes that represent the binary WASM for this module.

to_file(*f*)
Write this wasm module to file

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Definition**(**input*)
Base class for definition components.

A "definition" is a toplevel element in a WASM module that defines a type, function, import, etc.

class ppci.wasm.**Type**(**input*)
Defines the signature of a WASM function that is imported or defined in this module.

Flat form and abbreviations:

- In the flat form, a module has type definitions, and these are refered to with "type uses": `(type $xx)`.
- A type use can be given to *define* the type rather than reference it, this is resolved by the Module class.
- Multiple anonymous params may be combined: e.g. `(param i32 i32)`, this is resoved by this class, and to_string() applies this abbreviation.

Attributes:

- id: the id (str/int) of this definition in the type name/index space.
- params: a list of parameter tuples ($id, type), where id can be int.
- result: a list if type strings (0 or 1 elements in v1).

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Import**(**input*)
Import objects (from other wasm modules or from the host environment). Imports are handled at runtime by the host environment.

Flat form and abbreviations:

- Imports in flat form have a shape `(import "foo" "bar" ...)`.
- An import can be defined as func/table/memory/global that is "marked" as an import (e.g. `(memory (import "foo" "bar") 1)`. This is resolved by the Module class.

Attributes:

- modname: module to import from, as interpreted by host system.
- name: name of object to import, as interpreted by host system.
- kind: 'func', 'table', 'memory', or 'global'.
- id: the id to refer to the imported object.
- **info: a tuple who's content depends on the kind:**
  - func: (ref,) to the type (signature).
  - table: ('funcref', min, max), where max can be None.
  - memory: (min, max), where max can be None.
  - global: (typ, mutable)

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Table**(**input*)
A resizable typed array of references (e.g. to functions) that could not otherwise be stored as raw bytes in Memory (for safety and portability reasons). Only one default table can exist in v1.

A practical use-case is to store "function pointers" for e.g. callbacks. Tables allow doing that without actually exposing the memory location.

Flat form and abbreviations:

- Since v1 mandates a single table, the id is usually omitted.
- Elements can be specified inline, this is resolved by the Module class.
- The call_indirect instruction has one arg that specifies the signature, i.e. no support for inline typeuse.

Attributes:

- id: the id of this table definition in the table name/index space.
- kind: the kind of data stored in the table, only 'funcref' in v1.
- min: the minimum (initial) table size.
- max: the maximum table size, or None.

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Memory**(**input*)
Declares initial (and max) sizes of linear memory, expressed in WASM pages (64KiB). Only one default memory can exist in v1.

Flat form and abbreviations:

- Since v1 mandates a single memory, the id is usually omitted.
- Data can be specified inline, this is resolved by the Module class.

Attributes:

- id: the id of this memory definition in the memory name/index space.
- min: the minimum (initial) memory size.
- max: the maximum memory size, or None.

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Global**(**input*)
A global variable.

Attributes:

- id: the id of this global definition in the globals name/index space.
- typ: the value type of the global.
- mutable: whether this global is mutable (can hurt performance).
- init: an instruction to initialize the global (e.g. a i32.const).

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Export**(**input*)
Export an object defined in this module.

Flat form and abbreviations:

- Export in flat form have a shape `(export "foo" ...)`.

- An export can be defined as func/table/memory/global that is "marked" as an export (e.g. `(memory (export "bar") 1)`. This is resolved by the Module class.

Attributes:

- name: the name by which to export this value.
- kind: the kind of value to export ('func', 'table', or 'memory').
- ref: a reference to the thing being exported (in the name/index space corresponding to kind).

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Start**(**input*)
Define the index of the function to call at init-time. The func must have zero params and return values. There must be at most 1 start definition.

Attributes:

- ref: the reference to the function to mark as the start function.

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Func**(**input*)
The definition (i.e. instructions) of a function.

Flat form and abbreviations:

- In the flat form, it refers to a type (not define params inline).
- Inline signatures are resolved by ...
- Imported functions can be defined as e.g. `(func $add (import "foo" "bar"))`, which resolves into an Import instead of a Func.
- Multiple anonymous locals may be combined. This is resolved by this class and to_string() applies this abbreviation.

Attributes:

- id: the id of this func definition in the func name/index space.
- ref: the reference to the type (i.e. signature).
- locals: a list of ($id, typ) tuples. The id can be None to indicate implicit id's (note that the id is offset by the parameters).
- instructions: a list of instructions (may be given as tuples).

to_string()
Render function def as text

class ppci.wasm.**Elem**(**input*)
Define elements to populate a table.

Flat form and abbreviations:

- Elements can be defined inline inside Table expressions, this is resolved by the Module class.

Attributes:

- ref: the table id that this element applies to.
- offset: the element offset, expressed as an instruction list (i.e. [i32.const, end])
- refs: a list of function references.

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Data**(**input*)
Data to include in the module.

Flat form and abbreviations:

- Data can be defined inline inside Memory expressions, this is resolved by the Module class.

Attributes:

- ref: the memory id that this data applies to.
- offset: the byte offset, expressed as an instruction (i.e. i32.const)
- data: the binary data as a bytes object.

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Custom**(**input*)
Custom binary data.

to_string()
Get the textual representation (S-expression) of this component.

class ppci.wasm.**Ref**(*space*, *index=None*, *name=None*)
This is a reference to an object in one of the 5 spaces.

space must be one of 'type', 'func', 'memory', 'table', 'global', 'local' index can be none

is_zero
Check if we refer to element 0

ppci.wasm.**wasm_to_ir**(*wasm_module: ppci.wasm.components.Module*, *ptr_info*, *reporter=None*) → ppci.ir.Module
Convert a WASM module into a PPCI native module.

Parameters

- **wasm_module** (ppci.wasm.Module) – The wasm-module to compile
- **ptr_info** – ppci.arch.arch_info.TypeInfo size and alignment information for pointers.

Returns An IR-module.

ppci.wasm.**ir_to_wasm**(*ir_module: ppci.ir.Module*, *reporter=None*) → ppci.wasm.components.Module
Compiles ir-code to a wasm module.

Parameters

- **ir_module** (ir.Module) – The ir-module to compile
- **reporter** – optionally report compilation steps

Returns A wasm module.

class ppci.wasm.**WasmArchitecture**
Web assembly architecture description

ppci.wasm.**run_wasm_in_node**(*wasm*, *silent=False*)
Load a WASM module in node. Just make sure that your module has a main function.

ppci.wasm.**export_wasm_example**(*filename*, *code*, *wasm*, *main_js=''*)
Generate an html file for the given code and wasm module.

ppci.wasm.**run_wasm_in_notebook**(*wasm*)
Load a WASM module in the Jupyter notebook.

ppci.wasm.**has_node**
Check if nodejs is available

ppci.wasm.**instantiate**(*module*, *imports=None*, *target='native'*, *reporter=None*, *cache_file=None*)
Instantiate a wasm module.

Parameters

- **module** (ppci.wasm.Module) – The wasm-module to instantiate
- **imports** – A collection of functions available to the wasm module.
- **target** – Use 'native' to compile wasm to machine code. Use 'python' to generate python code. This option is slower but more reliable.
- **reporter** – A reporter which can record detailed compilation information.
- **cache_file** – a file to use as cache

ppci.wasm.**execute_wasm**(*module*, *args*, *target='python'*, *function=None*, *function_args=()*, *reporter=None*)
Execute the given wasm module.

ppci.wasm.**read_wasm**(*input*) → ppci.wasm.components.Module
Read wasm in the form of a string, tuple, bytes or file object. Returns a wasm Module object.

ppci.wasm.**read_wat**(*f*) → ppci.wasm.components.Module
Read wasm module from file handle

ppci.wasm.**wasmify**(*func*, *target='native'*)
Convert a Python function to a WASM function, compiled to native code. Assumes that all variables are floats. Can be used as a decorator, like Numba!

3.14 Utilities

3.14.1 leb128

Little Endian Base 128 (LEB128) variable length encoding

https://en.wikipedia.org/wiki/LEB128#Unsigned_LEB128

ppci.utils.leb128.**signed_leb128_decode**(*data*) → int
Read variable length encoded 128 bits signed integer.

```
>>> from ppci.utils.leb128 import signed_leb128_decode
>>> signed_leb128_decode(iter(bytes([0x9b, 0xf1, 0x59])))
-624485
```

ppci.utils.leb128.**signed_leb128_encode**(*value: int*) → bytes
Encode the given number as signed leb128

```
>>> from ppci.utils.leb128 import signed_leb128_encode
>>> signed_leb128_encode(-1337)
b'Çu'
```

ppci.utils.leb128.**unsigned_leb128_decode**(*data*) → int
Read variable length encoded 128 bits unsigned integer

```
>>> from ppci.utils.leb128 import unsigned_leb128_decode
>>> signed_leb128_decode(iter(bytes([0xe5, 0x8e, 0x26])))
624485
```

ppci.utils.leb128.**unsigned_leb128_encode**(*value: int*) → bytes
Encode number as into unsigned leb128 encoding

```
>>> from ppci.utils.leb128 import unsigned_leb128_encode
>>> unsigned_leb128_encode(42)
b'*'
```

3.14.2 Hexdump

Utilities to dump binary data in hex

ppci.utils.hexdump.**hexdump**(*data*, *address=0*, *width=16*)
Hexdump of the given bytes.

For example:

```
>>> from ppci.utils.hexdump import hexdump
>>> data = bytes(range(10))
>>> hexdump(data, width=4)
00000000  00 01 02 03  |....|
00000004  04 05 06 07  |....|
00000008  08 09        |..|
```

3.14.3 Codepage

Cool idea to load actual object code into memory and execute it from python using ctypes

Credits for idea: Luke Campagnola

class ppci.utils.codepage.**Mod**(*obj*, *imports=None*)
Container for machine code

get_symbol_offset(*name*)
Get the memory address of a symbol

ppci.utils.codepage.**load_code_as_module**(*source_file*, *reporter=None*)
Load c3 code as a module

ppci.utils.codepage.**load_obj**(*obj*, *imports=None*)
Load an object into memory.

Parameters

- **obj** – the code object to load.
- **imports** – A dictionary of functions to attach.

Optionally a dictionary of functions that must be imported can be provided.

3.14.4 Reporting

To create a nice report of what happened during compilation, this file implements several reporting types.

Reports can be written to plain text, or html.

class ppci.utils.reporting.**DummyReportGenerator**
Report generator which reports into the void

dump_exception(*einfo*)
List the given exception in report

dump_instructions(*instructions*, *arch*)
Print instructions

class ppci.utils.reporting.**HtmlReportGenerator**(*dump_file*)

annotate_source(*src*, *mod*)
Annotate source code.

dump_dag(*dags*)
Write selection dag to dumpfile

dump_exception(*einfo*)
List the given exception in report

dump_frame(*frame*)
Dump frame to file for debug purposes

dump_instructions(*instructions*, *arch*)
Print instructions

dump_raw_text(*text*)
Spitout text not to be formatted

dump_source(*name*, *source_code*)
Report web assembly module

tcell(*txt*, *indent=0*)
Inject a html table cell.

class ppci.utils.reporting.**MyHandler**(*backlog*)

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

class ppci.utils.reporting.**ReportGenerator**
Implement all these function to create a custom reporting generator

dump_exception(*einfo*)
List the given exception in report

dump_instructions(*instructions*, *arch*)
Print instructions

dump_wasm(*wasm_module*)
Report web assembly module

class ppci.utils.reporting.**TextReportGenerator**(*dump_file*)

dump_dag(*dags*)
Write selection dag to dumpfile

dump_exception(*einfo*)
List the given exception in report

dump_frame(*frame*)
Dump frame to file for debug purposes

class ppci.utils.reporting.**TextWritingReporter**(*dump_file*)

dump_instructions(*instructions*, *arch*)
Print instructions

print(**args*, *end='\n'*)
Convenience helper for printing to dumpfile

ppci.utils.reporting.**html_reporter**(*filename*)
Create a reporter that generates a HTML report.

3.15 Graph

This module contains some simple graph data structures and algorithms. Its by no means as extensive as networkx, but it has the things needed for some compilation algorithms.

3.15.1 Control flow graph

Reference

Control flow graph algorithms.

Functions present:

- dominators
- post dominators
- reachability
- dominator tree
- dominance frontier

class ppci.graph.cfg.**ControlFlowGraph**
Control flow graph.

Has methods to query properties of the control flow graph and its nodes.

Such as: - Dominators - Strict dominators - Immediate dominators - Post dominators - Strict post dominators - Immediate post dominators - Reachable nodes - Loops

bottom_up(*tree*)
Generator that yields all nodes in bottom up way

calculate_dominance_frontier()
Calculate the dominance frontier.

Algorithm from Ron Cytron et al.

how to calculate the dominance frontier for all nodes using the dominator tree.

calculate_loops()
Calculate loops by use of the dominator info

calculate_reach()
Calculate which nodes can reach what other nodes

children(*n*)
Return all children for node n

dominates(*one*, *other*)
Test whether a node dominates another node.

To test this, use the dominator tree, check where of the other node is below the one node in the tree by comparing discovery and finish intervals.

get_immediate_dominator(*node*)
Retrieve a nodes immediate dominator

get_immediate_post_dominator(*node*)
Retrieve a nodes immediate post dominator

post_dominates(*one*, *other*)
Test whether a node post dominates another node

strictly_dominates(*one*, *other*)
Test whether a node strictly dominates another node

validate()
Run some sanity checks on the control flow graph

class ppci.graph.cfg.**ControlFlowNode**(*graph*, *name=None*)

can_reach(*other*)
Test if this node can reach the another node

dominates(*other*)
Test whether this node dominates the other node

post_dominates(*other*)
Test whether this node post-dominates the other node

reached()
Test if this node is reached

class ppci.graph.cfg.**DomTreeNode**(*node*, *children*, *interval*)
A single node in the dominator tree.

below(*other*)
Test if this node is a descendant of this node.

below_or_same(*other*)
Test if this node is a descendant of this node (or is self)

class ppci.graph.cfg.**Loop**(*header*, *rest*)

header
Alias for field number 0

rest
Alias for field number 1

ppci.graph.cfg.**bottom_up**(*tree*)
Generator that yields all nodes in bottom up way

ppci.graph.cfg.**bottom_up_recursive**(*tree*)
Generator that yields all nodes in bottom up way

ppci.graph.cfg.**ir_function_to_graph**(*ir_function*)
Take an ir function and create a cfg of it

ppci.graph.cfg.**pre_order**(*tree*)
Traverse tree in pre-order

3.15.2 Graphs

The *ppci.graph* module can be used to work with graphs.

For example:

```
>>> from ppci.graph import Graph, Node
>>> g = Graph()
>>> n1 = Node(g)
>>> n2 = Node(g)
>>> n3 = Node(g)
>>> len(g)
3
>>> g.has_edge(n1, n2)
False
>>> n1.add_edge(n2)
>>> n1.add_edge(n3)
>>> g.has_edge(n1, n2)
```

(continues on next page)

(continued from previous page)

```
True
>>> n1.degree
2
>>> n2.degree
1
```

Reference

Graph package.

class ppci.graph.graph.**BaseGraph**
Base graph class

add_node(*node*)
Add a node to the graph

adjecent(*n*)
Return all unmasked nodes with edges to n

del_node(*node*)
Remove a node from the graph

get_degree(*node*)
Get the degree of a certain node

get_number_of_edges()
Get the number of edges in this graph

has_edge(*n*, *m*)
Test if there exist and edge between n and m

class ppci.graph.graph.**Graph**
Generic graph base class.

Can dump to graphviz dot format for example!

add_edge(*n*, *m*)
Add an edge between n and m

combine(*n*, *m*)
Merge nodes n and m into node n

del_edge(*n*, *m*)
Delete edge between n and m

del_node(*node*)
Remove a node from the graph

get_number_of_edges()
Get the number of edges in this graph

has_edge(*n*, *m*)
Test if there exist and edge between n and m

to_dot()
Render current graph to dot format

class ppci.graph.graph.**Node**(*graph*)
Node in a graph.

add_edge(*other*)
Create an edge to the other node

adjecent
Get adjecent nodes in the graph

degree
Get the degree of this node (the number of neighbours)

ppci.graph.graph.**topological_sort**(*nodes*)
Sort nodes topological, use Tarjan algorithm here See: https://en.wikipedia.org/wiki/Topological_sorting

Directed graph.

In a directed graph, the edges have a direction.

class ppci.graph.digraph.**DiGraph**
Directed graph.

add_edge(*n*, *m*)
Add a directed edge from n to m

del_edge(*n*, *m*)
Delete a directed edge

del_node(*node*)
Remove a node from the graph

get_number_of_edges()
Get the number of edges in this graph

has_edge(*n*, *m*)
Test if there exist and edge between n and m

predecessors(*node*)
Get the predecessors of the node

successors(*node*)
Get the successors of the node

class ppci.graph.digraph.**DiNode**(*graph*)
Node in a directed graph

predecessors
Get the predecessors of this node

successors
Get the successors of this node

ppci.graph.digraph.**dfs**(*start_node*, *reverse=False*)
Visit nodes in depth-first-search order.

Parameters

- **start_node** (–) – node to start with
- **reverse** (–) – traverse the graph by reversing the edge directions.

Dominators in graphs are handy informations.

Lengauer and Tarjan developed a fast algorithm to calculate dominators from a graph.

Algorithm 19.9 and 19.10 as can be found on page 448 of Appel.

class ppci.graph.lt.**LengauerTarjan**(*reverse*)
The lengauer Tarjan algorithm for calculating dominators

ancestor_with_lowest_semi(*v*)
O(log N) implementation with path compression.

Rewritten from recursive function to prevent hitting the recursion limit for large graphs.

ancestor_with_lowest_semi_fast(*v*)
The O(log N) implementation with path compression.

This version suffers from a recursion limit for large graphs.

ancestor_with_lowest_semi_naive(*v*)
O(N^2) implementation.

This is a slow algorithm, path compression can be used to increase speed.

dfs(*start_node*)
Depth first search nodes

link(*p*, *n*)
Mark p as parent from n

3.15.3 Finding loops

To detect program structure like loops and if statements from basic blocks, one can use algorithms in these classes to find them.

In the below example an ir function is defined, and then the loops are detected.

```
>>> import io
>>> from ppci.graph.relooper import find_structure, print_shape
>>> from ppci.irutils import read_module, verify_module
>>> ir_source = """
... module demo;
... global function i32 inc(i32 a, i32 b) {
...   inc_block0: {
...     jmp inc_block1;
...   }
...   inc_block1: {
...     i32 x = phi inc_block0: a, inc_block1: result;
...     i32 result = x + b;
...     cjmp result > b ? inc_block2 : inc_block1;
...   }
...   inc_block2: {
...     return result;
...   }
... }
... """
>>> ir_module = read_module(io.StringIO(ir_source))
>>> verify_module(ir_module)
>>> ir_module.stats()
'functions: 1, blocks: 3, instructions: 5'
>>> ir_function = ir_module.get_function('inc')
>>> shape, _ = find_structure(ir_function)
>>> print_shape(shape)
   code: CFG-node(inc_block0)
   loop
      if-then CFG-node(inc_block1)
         code: CFG-node(inc_block2)
      else
         Continue-shape 0
      end-if
   end-loop
```

As can be seen, the program contains one loop.

Reference

Rverse engineer the structured control flow of a program.

Possible algorithms:

References:

- relooper https://github.com/kripken/Relooper/blob/master/paper.pdf

[Cifuentes1998]

[Baker1977]

A hint might be "Hammock graphs" which are single entry, single exit graphs. They might be the join point between dominator and post dominator nodes.

The algorithm for finding a program structure is as following:

- Create a control flow graph from the ir-function.
- Find loops in the control flow graph
- **Now start with entry node, and check if this node is:**
  - a start of a loop
  - an if statement with two outgoing control flow paths
  - straight line code

class ppci.graph.relooper.**BasicShape**(*content*)

class ppci.graph.relooper.**BreakShape**(*level*)

class ppci.graph.relooper.**ContinueShape**(*level*)

class ppci.graph.relooper.**IfShape**(*content*, *yes_shape*, *no_shape*)
If statement

class ppci.graph.relooper.**LoopShape**(*body*)
Loop shape

class ppci.graph.relooper.**MultipleShape**
Can be a switch statement?

class ppci.graph.relooper.**Relooper**
Implementation of the relooper algorithm

class ppci.graph.relooper.**SequenceShape**(*shapes*)

class ppci.graph.relooper.**Shape**
A control flow shape.

ppci.graph.relooper.**find_structure**(*ir_function*)
Figure out the block structure of

Parameters **ir_function** – an *ir.SubRoutine* containing a soup-of-blocks.

Returns A control flow tree structure.

3.15.4 Calltree

This module can be used to determine the calltree of a program. This can be helpful in understanding which functions calls which other function.

A callgraph is a graph of functions which call eachother.

class ppci.graph.callgraph.**CallGraph**

ppci.graph.callgraph.**mod_to_call_graph**(*ir_module*) → ppci.graph.callgraph.CallGraph
Create a call graph for an ir-module

Calculate the cyclomatic complexity.

Cyclomatic complexity is defined as:

C = E - N + 2 * P

Where: E = number of edges N = number of nodes P = number of connected components.

For functions and procedures P = 1.

So the formula for a single subroutine is:

C = E - N + 2

ppci.graph.cyclo.**cyclomatic_complexity**(*cfg*)
Determine the cyclomatic complexity of a CFG (control-flow-graph)

3.16 Links

This section contains links and resources to other projects, papers and books.

3.16.1 Classical compilers

The following list gives a summary of some compilers that exist in the open source land.

- *LLVM*

  A relatively new and popular compiler. LLVM stands for low level virtual machine, a compiler written in C++. This is a big inspiration place for ppci! http://llvm.org

- *GCC*

  The gnu compiler. The famous open source compiler. https://gcc.gnu.org/

3.16.2 Other compilers

- *ACK*

  The amsterdam compiler kit. http://tack.sourceforge.net/

- *lcc*

  A retargetable C compiler. https://github.com/drh/lcc

- *sdcc*

  The small device c compiler. http://sdcc.sourceforge.net/

- *8cc*

  A small C compiler in C99. https://github.com/rui314/8cc

3.16.3 Other compilers written in python

- *zxbasic*

  Is a freebasic compiler written in python. http://www.boriel.com/wiki/en/index.php/ZXBasic

- *python-msp430-tools*

  A msp430 tools project in python. https://launchpad.net/python-msp430-tools

- *peachpy*

  A x86 64 assembly code generator. https://github.com/Maratyszcza/PeachPy

- *pycca*

  An x86 assembly code generator and compiler in python. https://github.com/campagnola/pycca

- *llvm-codegen-py*

  Machine code generation in python from llvm ir code. https://github.com/pfalcon/llvm-codegen-py

- *textx*

  A toolkit to create DSL tools. https://github.com/igordejanovic/textX
- amoco

  Static analysis of binaries. https://github.com/bdcht/amoco
- pythran

  http://pythran.readthedocs.io/en/latest/
- Shivyc - A C compiler implemented in python

  https://github.com/ShivamSarodia/ShivyC

3.16.4 Other C-related tools written in python

pycparser

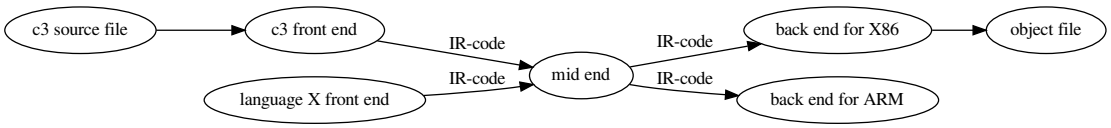
3.16.5 Citations

CHAPTER 4

Compiler internals

This chapter describes the design of the compiler. The compiler consists a frontend, mid-end and back-end. The frontend deals with source file parsing and semantics checking. The mid-end performs optimizations. This is optional. The back-end generates machine code. The front-end produces intermediate code. This is a simple representation of the source. The back-end can accept this kind of representation.

The compiler is greatly influenced by the LLVM design.



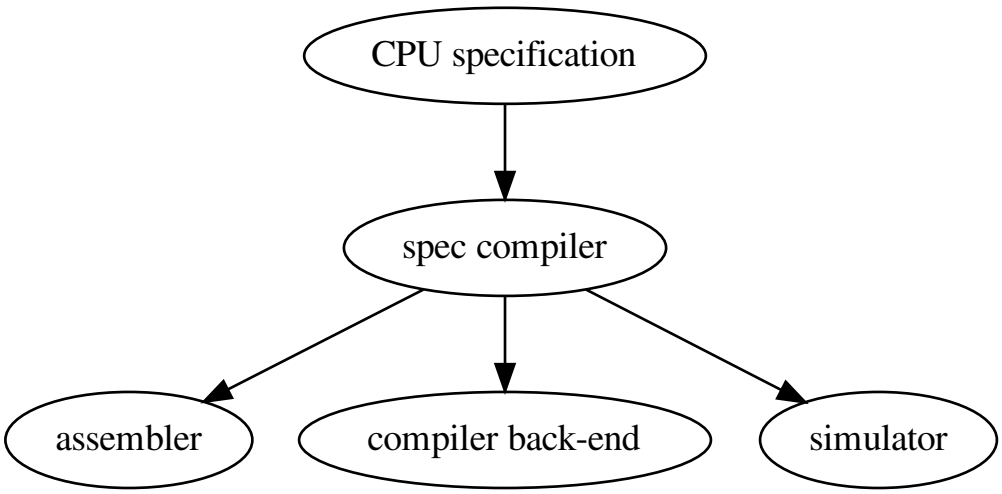
4.1 Specification languages

4.1.1 Introduction

DRY

Do not repeat yourself (DRY). This is perhaps the most important idea to keep in mind when writing tools like assemblers, disassemblers, linkers, debuggers and compiler code generators. Writing these tools can be a repetitive and error prone task.

One way to achieve this is to to write a specification file for a specific processor and generate from this file the different tools. The goal of a machine description file is to describe various aspects of a CPU architecture and generate from it tools like assemblers, disassemblers, linkers, debuggers and simulators.

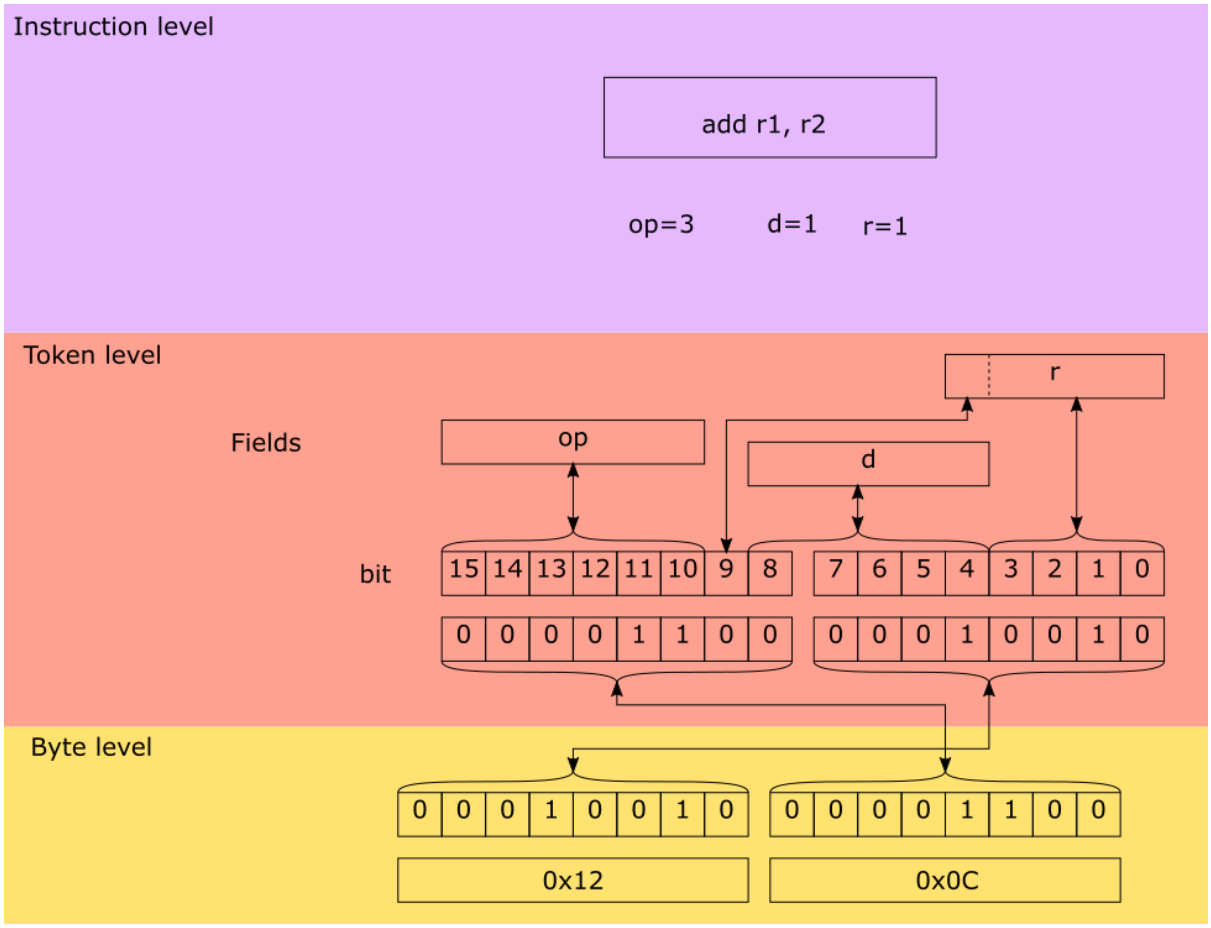


4.1.2 Design

The following information must be captured in the specification file:

- Assembly textual representation
- Binary representation
- Link relocations
- Mapping from compiler back-end
- Effects of instruction (semantics)

The following image depicts the encoding and decoding of the AVR add instruction.



The following code demonstrates how this instruction is described.

First the proper token is defined:

```
from ppci.arch.token import Token, bit, bit_range, bit_concat

class AvrArithmaticToken(Token):
    class Info:
        size = 16

    op = bit_range(10, 16)
    r = bit_concat(bit(9), bit_range(0, 4))
    d = bit_range(4, 9)
```

Then the instruction is defined, defining a syntax and the mapping of token fields to instruction parameters:

```
from ppci.arch.avr.registers import AvrRegister
from ppci.arch.encoding import Instruction, Operand, Syntax

class Add(Instruction):
    tokens = [AvrArithmaticToken]
    rd = Operand('rd', AvrRegister, read=True, write=True)
    rr = Operand('rr', AvrRegister, read=True)
    syntax = Syntax(['add', ' ', rd, ',', ' ', rr])
    patterns = {'op': 0b11, 'r': rr, 'd': rd}
```

```
>>> from ppci.arch.avr import registers
>>> a1 = Add(registers.r1, registers.r2)
>>> str(a1)
'add r1, r2'
```

(continues on next page)

(continued from previous page)

```
>>> a1.encode()
b'\x12\x0c'
```

4.1.3 Background

There are several existing languages to describe machines in a Domain Specific Language (DSL). Examples of these are:

- Tablegen (llvm)
- cgen (gnu)
- LISA (Aachen)
- nML (Berlin)
- SLED (Specifying representations of machine instructions (Norman Ramsey and Mary F. Fernandez))

http://www.cs.tufts.edu/~nr/toolkit/

Concepts to use in this language:

- Single stream of instructions
- State stored in memory
- Pipelining
- Instruction semantics

Optionally a description in terms of compiler code generation can be attached to this. But perhaps this clutters the description too much and we need to put it elsewhere.

The description language can help to expand these descriptions by expanding the permutations.

4.1.4 Example specifications

For a complete overview of ADL (Architecture Description Language) see [overview].

llvm

```
def IMUL64rr : RI<0xAF, MRMSrcReg, (outs GR64:$dst),
                                   (ins GR64:$src1, GR64:$src2),
                    "imul{q}\t{$src2, $dst|$dst, $src2}",
                    [(set GR64:$dst, EFLAGS,
                        (X86smul_flag GR64:$src1, GR64:$src2))],
                    IIC_IMUL64_RR>,
                TB;
```

LISA

```
<insn> BC
{
  <decode>
  {
    %ID: {0x7495, 0x0483}
    %cond_code: { %OPCODE1 & 0x7F }
    %dest_address: { %OPCODE2 }
  }
```

(continues on next page)

(continued from previous page)

```
  <schedule>
  {
    BC1(PF, w:ebus_addr, w:pc) |
    BC2(PF, w:pc), BC3(IF) |
    BC4(ID) |
    <if> (condition[cond_code])
    {
      BC5(AC) |
      BC6(PF), BC7(ID), BC8(RE) |
      BC9(EX)
    }
    <else>
    {
      k:NOP(IF), BC10(AC, w:pc) |
      BC11(PF), BC12(ID), BC13(RE) |
      k:NOP(ID), BC14(EX) |
      k:NOP(ID), k:NOP(AC) |
      k:NOP(AC), k:NOP(RE) |
      k:NOP(RE), k:NOP(EX) |
      k:NOP(EX)
    }
  }
  <operate>
  {
    BC1.control: { ebus_addr = pc++; }
    BC2.control: { ir = mem[ebus_addr]; pc++ }
    BC10.control: { pc = (%OPCODE2) }
  }
}
```

SLED

```
patterns
  nullary is any of [ HALT NEG COM SHL SHR READ WRT NEWL NOOP TRA NOTR ],
    which is op = 0 & adr = { 0 to 10 }
constructors
  IMULb        Eaddr            is      (grp3.Eb;    Eaddr) & IMUL.AL.eAX
```

nML

```
type word = card(16)
type absa = card(9)
type disp = int(4)
type off = int(6)
mem PC[1,word]
mem R[16,word]
mem M[65536,word]
var L1[1,word]
var L2[1,word]
var L3[1,word]
mode register(i:card(4)) = R[i]
  syntax = format("R%s", i)
  image = format("%4b", i)
mode memory = ind | post | abs
mode ind(r:register, d:disp) = M[r+d]
  update = {}
  syntax = format("@%s(%d)", r.syntax, d)
```

(continues on next page)

(continued from previous page)

```
  image = format("0%4b%4b0", r.image, d)
mode post(r:register, d:disp) = M[r+d]
  update = { r = r + 1; }
  syntax = format("@%s++(%d)", r.syntax, d)
  image = format("0%4b%4b1", r.image, d)
mode abs(a : absa) = M[a]
  update = {}
  syntax = format("%d", a)
  image = format("1%9b", a)
op instruction( i : instr )
  syntax = i.syntax
  image = i.image
  action = {
    PC = PC + 1;
    i.action;
  }
op instr = move | alu | jump
op move(lore:card(1), r:register, m:memory)
  syntax = format("MOVE%d %s %s", lore, r.syntax, m.syntax)
  image = format("0%1b%4b%10b", lore, r.image, m.image)
  action = {
    if ( lore ) then r = m;
    else m = r;
    endif;
    m.update;
  }
op alu(s1:register, s2:register, d:reg, a:aluop)
  syntax = format("%s %s %s %s", a.syntax, s1.syntax, s2.syntax, d.syntax)
  image = format("10%4b%4b%4b%2b", s1.image, s2.image, d.image, a.image)
  action = {
    L1 = s1; L2 = s2; a.action; d = L3;
  }
op jump(s1:register, s2:register, o:off)
  syntax = format("JUMP %s %s %d", s1.syntax, s2.syntax, o)
  image = format("11%4b%4b%6b", s1.image, s2.image, o)
  action = {
   if ( s1 >= S2 ) then PC = PC + o;
   endif;
  }
op aluop = and | add | sub | shift;
op and() syntax = "and" image = "00" action = { L3 = L1 & L2; }
op add() syntax = "add" image = "10" action = { L3 = L1 + L2; }
op sub() syntax = "sub" image = "01" action = { L3 = L1 - L2; }
```

4.2 Hardware description classes

On this page the different terms used when dealing with hardware are explained.

core A cpu can have one or more cores.

A **cpu** is the central processing unit of a chip.

machine

An **architecture** can be more or less seen as a family of cpu's. Most cpus of a certain architecture have the same ISA.

An **isa** is the instruction set architecture. This means basically the set of instructions that are available. A cpu is said to implement a certain ISA.

4.2.1 PC

A **system** is one of linux, windows, mac, helenos, minix, qnx, freebsd or another system.

4.2.2 Embedded

A **chip** is a specific package which can be a microcontroller having a cpu or more then one.

A **board** is a specific PCB, usually containing a cpu or more.

target

4.3 IR-code

The intermediate representation (IR) of a program de-couples the front end from the backend of the compiler.

See *IR-code* for details about all the available instructions.

CHAPTER 5

Faq

Why did you do this?

There are several reasons:

- it is possible! (For me, this already is a sufficient explanation :))
- this compiler is very portable due to python being portable.
- writing a compiler is a very challenging task

Is this compiler slower than compilers written in C/C++?

Yes. Although a comparison is not yet done, this will be the case, due to the overhead and slower execution of python code.

Cool project, I want to contribute to this project, what can I do?

Great! Please *see this page*. If you are not sure where to begin, please contact me first. For a list of tasks, refer to *the todo page*. For hints on development see *the development page*.

Why would one want to make a compiler for such a weird language such as C?

Because of the enormous amount of C-source code available. This serves as a great test suite for the compiler.

Why are the so many different parts in ppci, should you not better split this package into several smaller ones?

This would certainly be possible, but for now I would like to keep all code in a single package. This simplifies the dependencies, and also ensures that a single version works as a whole. Another benefit is that the packaging overhead is reduced. Also, functionality can be easily shared (but should not lead to spaghetti code of course).

CHAPTER 6

Contributing

This section is intended for contributors of ppci.

6.1 Support

Ppci is still in development, and all help is welcome!

6.1.1 How to help

If you want to add some code to the project, the best way is to create a fork of the project, make your changes and create a pull request. You can also help to improve the documentation if you find something that is not clear.

For a list of tasks, refer to *the todo page*. For help on getting started with development see *the development page*.

6.1.2 How to submit a patch

This section explains how to add something to ppci.

1. Clone the repository at bitbucket to your own account.
2. Communicate via e-mail / chat the change you wish to make.
3. Create a hg bookmark, not a branch, and make your change. Make sure you use the default branch.
4. Create a pull request.

6.1.3 Report an issue

If you find a bug, you can file it here:

https://github.com/windelbouwman/ppci/issues

6.2 Communication

Join the #ppci irc channel on freenode!

Or visit the forum:

- https://groups.google.com/d/forum/ppci-dev

Or chat on gitter:

- https://gitter.im/ppci-chat/Lobby

6.3 Development

This chapter descibes how to develop on ppci.

6.3.1 Source code

The sourcecode repository of the project is located at these locations:

- https://mercurial.tuxfamily.org/ppci/ppci
- https://github.com/windelbouwman/ppci
- https://pikacode.com/windel/ppci/

To check out the latest code and work use the development version use these commands to checkout the source code and setup ppci such that you can use it without having to setup your python path.

```
$ mkdir HG
$ cd HG
$ hg clone https://mercurial.tuxfamily.org/ppci/ppci
$ cd ppci
$ sudo python setup.py develop
```

6.3.2 Coding style

All code is intended to be pep8 compliant. You can use the pep8 tool, or run:

```
$ tox -e flake8
```

This will check the code for pep8 violations.

On top of this, we use the black formatter to autoformat code.

Future work includes using pylint and mypy for more static code analysis.

6.3.3 Running the testsuite

To run the unit tests with the compiler, use pytest:

```
$ python -m pytest -v test/
```

Or, yet another way, use tox:

```
$ tox -e py3
```

In order to test ppci versus different versions of python, tox is used. To run tox, simply run in the root directory:

```
$ tox
```

Note that those command will **not work properly**:

```
$ python -m unittest discover -s  # will not recursively discover test cases
$ python setup.py test  # does not work and is deprecated
```

Read more about testing.

Profiling

If some part is slow, it can be handy to run a profiler. To do this, run the slow script with the cProfile. The output can be viewed with pyprof2calltree.

```
$ python -m cProfile -o profiled.out slow_script.py
$ pip install pyprof2calltree
$ pyprof2calltree -i profiled.out -k
```

6.3.4 Building the docs

The docs can be built locally by using sphinx. Sphinx can be invoked directly:

```
$ cd docs
$ sphinx-build -b html . build
```

Alternatively the tox docs environment can be used:

```
$ tox -e docs
```

6.3.5 Directory structure

- ppci : source code of the ppci library
  - arch : different machine support
    - arm : arm support
    - avr : avr support
    - microblaze
    - mips
    - msp430 : msp430 support
    - riscv
    - stm8
    - x86_64
    - xtensa : xtensa support
  - binutils : assembler and linker
  - cli : command line interface utilities
  - codegen : code generation
  - format : various file formats
  - lang : human readable languages
    - c : c frontend

        * c3 : c3 frontend
        * python : python compilation code
        * tools : language tools
    - opt : IR-code optimization
    - util : utilities
- docs : documentation
- examples : directory with example projects
- test : tests

6.3.6 Continuous integration

The compiler is tested for linux:

- https://travis-ci.org/windelbouwman/ppci-mirror

and for windows:

- https://ci.appveyor.com/project/WindelBouwman/ppci-786

6.3.7 Code metrics

Code coverage is reported to the codecov service:

- https://codecov.io/bb/windel/ppci/branch/default

Other code metrics are listed here:

- https://www.openhub.net/p/ppci
- https://libraries.io/pypi/ppci

6.4 Debugging

6.4.1 Debugging tests

To debug test cases, a handy trick is to use pudb (when not using fancy ide like vscode or pycharm). To do this, specify the debugger to use with pytest like this:

```
$ pytest -v --pdb --pdbcls pudb.debugger:Debugger --capture=no
```

6.4.2 Debugging dynamic code

Sometimes, the python interpreter might crash due to playing with dynamically injected code. To debug this, we can use gdb for example.

```
$ gdb --args python script.py
(gdb) run
```

Once the program crashes, one can disassemble and print info:

```
(gdb) bt
(gdb) disassemble /r 0x7fff000, 0x7fff200
(gdb) info registers
```

6.4.3 Debugging python code

Alternatively, when facing a python exception, one might want to try the pudb debugger like this:

```
$ python -m pudb crashing_script.py
```

6.4.4 Debugging sample snippets

The folder `test/samples` contains sample snippets of code with corresponding output. Those samples are compiled and run during testing. If one of those samples fails, troubleshooting begins.

There are several approaches to pinpoint the issue:

- Examination of the compilation report. Each test case generates a HTML report with detailed information about the compilation steps. This can be handy to determine where the process generated wrong code.
- Run the sample binary with QEMU. This involves stepping through the generated binary and keeping an eye on register and memory values.

6.4.5 Debugging with QEMU

Qemu has a mode in which it will dump very detailed information about each instruction executed.

Use the following line to get a whole lot of info:

```
$ qemu-system-or1k -D trace.txt -d in_asm,exec,int,op_opt,cpu -singlestep
```

Note the -singlestep option here will execute every single instruction at a time.

This can produce helpful instruction tracing like this:

```
----------------
IN:
0x00000100:  l.j       2

OP after optimization and liveness analysis:
ld_i32 tmp0,env,$0xfffffffffffffff0      dead: 1  pref=0xffff
movi_i32 tmp1,$0x0                       pref=0xffff
brcond_i32 tmp0,tmp1,lt,$L0              dead: 0 1

---- 00000100 00000000
movi_i32 jmp_pc,$0x108                   sync: 0  dead: 0  pref=0xffff
movi_i32 dflag,$0x1                      sync: 0  dead: 0  pref=0xffff
movi_i32 ppc,$0x100                      sync: 0  dead: 0  pref=0xffff
goto_tb $0x0
movi_i32 pc,$0x104                       sync: 0  dead: 0  pref=0xffff
exit_tb $0x7fc8a8300040
set_label $L0
exit_tb $0x7fc8a8300043

Trace 0: 0x7fc8a8300100 [00000000/00000100/0x5]
PC=00000100
R00=00000000 R01=00000000 R02=00000000 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=00000000
R16=00000000 R17=00000000 R18=00000000 R19=00000000
R20=00000000 R21=00000000 R22=00000000 R23=00000000
R24=00000000 R25=00000000 R26=00000000 R27=00000000
R28=00000000 R29=00000000 R30=00000000 R31=00000000
----------------
```

(continues on next page)

(continued from previous page)

```
IN:
0x00000104:  l.nop

OP after optimization and liveness analysis:
ld_i32 tmp0,env,$0xfffffffffffffff0      dead: 1  pref=0xffff
movi_i32 tmp1,$0x0                       pref=0xffff
brcond_i32 tmp0,tmp1,lt,$L0              dead: 0 1

---- 00000104 00000001
movi_i32 dflag,$0x0                      sync: 0  dead: 0  pref=0xffff
movi_i32 ppc,$0x104                      sync: 0  dead: 0  pref=0xffff
mov_i32 pc,jmp_pc                        sync: 0  dead: 0 1  pref=0xffff
discard jmp_pc                           pref=none
call lookup_tb_ptr,$0x6,$1,tmp2,env      dead: 1  pref=none
goto_ptr tmp2                            dead: 0
set_label $L0
exit_tb $0x7fc8a8300183

Linking TBs 0x7fc8a8300100 [00000100] index 0 -> 0x7fc8a8300240 [00000104]
Trace 0: 0x7fc8a8300240 [00000000/00000104/0x7]
PC=00000104
R00=00000000 R01=00000000 R02=00000000 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=00000000
R16=00000000 R17=00000000 R18=00000000 R19=00000000
R20=00000000 R21=00000000 R22=00000000 R23=00000000
R24=00000000 R25=00000000 R26=00000000 R27=00000000
R28=00000000 R29=00000000 R30=00000000 R31=00000000
----------------
```

6.5 Testing

6.5.1 Long tests

There are a series of test snippets located in the test/samples folder. If you want to run these, you can use this:

```
$ LONGTESTS=all python -m pytest test/
```

Valid values for the LONGTESTS variable (note that multiple values can be seperated by comma's):

value	meaning
all	Go all in, run all tests possible
any	Run some extra tests which take somewhat longer
python	Convert sample snippets to python
jit	Run a tricky test in which code is jitted
wasm	Convert sample snippets to wasm
riscv,msp430, avr,microblaze, xtensa,arm	Convert sample snippets to code for the given architecture

Some targets need iverilog to emulate a certain processor. If you want to run these, use this:

```
$ LONGTESTS=all IVERILOG=1 python -m pytest test/
```

6.5.2 3rd party test suites

There exist many different compiler validation suites. Some of them are pure validation sets, others are part of a compiler toolchain. In order to use these test suites, a series of test suite adapter files exists.

Available test adapters:

- mcpp (set *MCPP_DIR*) *test/lang/c/test_mcpp_test_suite.py*
- fortran compiler validation system 2.1 (set *FCVS_DIR*) *test/lang/fortran/test_fortran_test_suite.py*

WebAssembly spec

The WebAssembly specification contains a validation suite.

To use these tests, clone https://github.com/WebAssembly/spec and set the environment variable WASM_SPEC_DIR to the location where the code was cloned.

To run the test spec tests:

```
$ export WASM_SPEC_DIR=~/GIT/spec
$ python -m pytest test/wasm/test_suite_full -v
```

C testsuite

The c-testsuite is a collection of C test cases.

Usage with pytest:

> $ export C_TEST_SUITE_DIR=/path/to/GIT/c-testsuite $ python -m pytest test/lang/c/test_c_test_suite.py -v

Usage as a script:

> $ python test_c_test_suite.py /path/to/GIT/c-testsuite

See also:

https://github.com/c-testsuite/c-testsuite

6.5.3 Compiler testing

There are a number of ways to stress test the compiler.

One way is to compile existing C sourcecode, and see if the code compiles and runs.

Current results:

test sample	Compiles	Runs
libmad	yes	?
8cc	yes	?
lcc	no	?
micropython	no	?

libmad

The sourcecode for this test can be found here: https://www.underbit.com/products/mad/

To compile libmad, use the script *tools/compile_libmad.py*. This will compile the libmad sourcecode.

Compilation takes 45 seconds.

8cc

8cc is a small C compiler which can compile itself. PPCI can also compile it, running it remains a challenge.

Sourcecode is located here: https://github.com/rui314/8cc

To compile 8cc, use the script *tools/compile_8cc.py*

lcc

lcc is a C compiler written in C. Sourcecode can be found here: https://github.com/drh/lcc

To compile this sourcecode, use the script *tools/compile_lcc.py*

micropython

Micropython is a python implementation for microcontrollers. Website: http://micropython.org/

To compile this sourcecode, use the script *tools/compile_micropython.py*

6.6 Todo

Below is a list of features / tasks that need to be done.

- Improve the debugger.
  - Add support for local variables.
- Misc
  - Create a better/fancier logo
  - Improve the fuzzer tool that can generate random source code to stress the compiler.
- Binary utils
  - Implement the disassembler further.
  - Implement Mac OSX support and add a mac 64-bit example project.
  - Add Windows support and add a windows 64-bit example project.
- Languages
  - Complete the fortran frontend. The ppci.lang.fortran module contains a start.
  - Complete the C frontend, The ppci.lang.c module contains an attempt.
  - Complete the pascal frontend.
  - Complete the front-end for LLVM IR-code, this way, the front-end of LLVM can be used and the backend of ppci.
  - Add a backend for LLVM IR-code.
  - Investigate regular expression derivatives as described here: https://github.com/MichaelPaddon/epsilon
  - Add ada frontend
  - Add D language frontend
- Optimizations
  - Add a peephole optimizer.
  - Investigate polyhedral optimization

- Add better support for harvard architecture cpu's like avr, 8051 and PIC.

6.6.1 Issue trackers

On github, there is a list of open issues which need attention:

https://github.com/windelbouwman/ppci/issues

6.7 Release procedure

This is more a note to self section on how to create a new release.

1. Determine the version numbers of this release and the next.
2. Switch to the release branch and merge the default branch into the release branch.

```
$ hg update release
$ hg merge default
$ hg commit
```

3. Check the version number in ppci/__init__.py
4. Make sure all tests pass and fix them if not.

```
$ tox
```

5. Tag this release with the intended version number and update to this tag.

```
$ hg tag x.y.z
$ hg update x.y.z
```

6. Package and upload the python package. The following command creates a tar gz archive as well as a wheel package.

```
$ python setup.py sdist bdist_wheel upload
```

7. Switch back to the default branch and merge the release branch into the default branch.

```
$ hg update default
$ hg merge release
$ hg commit
```

8. Increase the version number in ppci/__init__.py.
9. Update docs/changelog.rst

6.8 Contributors

Ppci is brought to you by the following people:

- Windel, maintainer
- Michael, riscv architecture and gdb debugging client
- Henk, stm8 architecture
- Almar, Addition of wasm and program api

CHAPTER 7

Changelog

7.1 Release 1.0 (Planned)

- platform.python_compiler() returns 'ppci 1.0'

7.2 Release 0.5.9 (Upcoming)

7.3 Release 0.5.8 (Jun 8, 2020)

- Add inline asm support to C frontend.
- Work on pascal frontend
- Add initial support for relocatable ELF file
- Add initial command to run wasm files directly
- CoreMark benchmark can be compiled and run

7.4 Release 0.5.7 (Dec 31, 2019)

- Added support for the m68k processor.
- Added support for the microblaze processor.
- Added RISCV floating point support (by Michael).
- Add S-record format.
- Add amiga hunk format.
- Add OCaml bytecode file reader.
- Use black formatter

7.5 Release 0.5.6 (Aug 22, 2018)

- Compilation is now reproducible: recompiling the same source yields exactly the same machine code
- Added support for webassembly WAT file format (by Almar)
- Improved compilation speed
- Separation of the graph algorithms in ppci.graph.

7.6 Release 0.5.5 (Jan 17, 2018)

- Addition of WASM support (by Almar)
- Added compilation of subset of python
- Changed commandline tools from scripts into setuptools entrypoints
- Add support for function pointers in C
- Add uboot legacy image utility
- Start of exe support
- Start of mips support

7.7 Release 0.5.4 (Aug 26, 2017)

- Addition of open risc (or1k) architecture support
- Added command line options to emit assembly output
- Created ppci.lang.tools module with helper classes for parsing and lexing

7.8 Release 0.5.3 (Apr 27, 2017)

- Initial version of C preprocessor
- Improved calling convention handling
- Initial version of pascal front-end

7.9 Release 0.5.2 (Dec 29, 2016)

- Better floating point support in c3
- Addition of the xtensa target architecture
- Extended the supported 6502 instructions

7.10 Release 0.5.1 (Oct 16, 2016)

- Expand the riscv example to include single stepping (by Michael)
- Bugfix in byte parameter passing for x86_64 target
- Cleanup of the encoding system

- Start with llvm-IR frontend

7.11 Release 0.5 (Aug 6, 2016)

- Debug type information stored in better format
- Expression evaluation in debugger
- Global variables can be viewed
- Improved support for different register classes

7.12 Release 0.4.0 (Apr 27, 2016)

- Start with debugger and disassembler

7.13 Release 0.3.0 (Feb 23, 2016)

- Added risc v architecture
- Moved thumb into arm arch
- msp430 improvements

7.14 Release 0.2.0 (Jan 23, 2016)

- Added linker (ppci-ld.py) command
- Rename *buildfunctions* to *api*
- Rename *target* to *arch*

7.15 Release 0.1.0 (Dec 29, 2015)

- Added x86_64 target.
- Added msp430 target.

7.16 Release 0.0.5 (Mar 21, 2015)

- Remove st-link and hence pyusb dependency.
- Support for pypy3.

7.17 Release 0.0.4 (Feb 24, 2015)

7.18 Release 0.0.3 (Feb 17, 2015)

7.19 Release 0.0.2 (Nov 9, 2014)

7.20 Release 0.0.1 (Oct 10, 2014)

- Initial release.

Bibliography

[Smith2004] "A generalized algorithm for graph-coloring register allocation", 2004, Michael D. Smith and Norman Ramsey and Glenn Holloway.

[Runeson2003] "Retargetable Graph-Coloring Register Allocation for Irregular Architectures", 2003, Johan Runeson and Sven-Olof Nystrom. http://user.it.uu.se/~svenolof/wpo/AllocSCOPES2003.pdf

[George1996] "Iterated Register Coalescing", 1996, Lal George and Andrew W. Appel.

[Briggs1994] "Improvements to graph coloring register allocation", 1994, Preston Briggs, Keith D. Cooper and Linda Torczon

[Chaitin1982] "Register Allocation and Spilling via Graph Coloring", 1982, G. J. Chaitin.

[Cifuentes1998] "Assembly to High-Level Language Translation", 1998, Cristina Cifuentes and Doug Simon and Antoine Fraboulet.

[Davidson1980] "The Design and Application of a Retargetable Peephole Optimizer", 1980, Jack W. Davidson and Christopher W. Fraser.

[Baker1977] "An Algorithm for Structuring Flowgraphs", 1977, B.S. Baker. https://dl.acm.org/citation.cfm?id=321999

[overview] http://esl.cise.ufl.edu/Publications/iee05.pdf

Python Module Index

W

Index

Symbols

ppci-asm command line option, 34
ppci-build command line option, 33
ppci-c3c command line option, 32
ppci-cc command line option, 38
ppci-hexdump command line option, 51
ppci-java command line option, 49
ppci-ld command line option, 35
ppci-objcopy command line option, 36
ppci-objdump command line option, 37
ppci-ocaml command line option, 47
ppci-opt command line option, 37
ppci-pascal command line option, 40
ppci-pycompile command line option, 41
ppci-readelf command line option, 42
ppci-wabt command line option, 46
ppci-wasm2wat command line option, 45
ppci-wasmcompile command line option, 43
ppci-wat2wasm command line option, 45
ppci-yacc command line option, 44
-segment , -S
ppci-objcopy command line option, 36
-std {c89,c99}
ppci-cc command line option, 39
-super-verbose
ppci-cc command line option, 39
-target {native,python}
ppci-wabt-run command line option, 46
-text-report
ppci-archive command line option, 33
ppci-asm command line option, 34
ppci-build command line option, 33
ppci-c3c command line option, 32
ppci-cc command line option, 38
ppci-hexdump command line option, 51
ppci-java command line option, 49
ppci-ld command line option, 35
ppci-objcopy command line option, 36
ppci-objdump command line option, 37
ppci-ocaml command line option, 48
ppci-opt command line option, 37
ppci-pascal command line option, 40
ppci-pycompile command line option, 41
ppci-readelf command line option, 42
ppci-wabt command line option, 46
ppci-wasm2wat command line option, 45
ppci-wasmcompile command line option, 43
ppci-wat2wasm command line option, 45
ppci-yacc command line option, 44
-trigraphs
ppci-cc command line option, 39
-verbose, -v
ppci-archive command line option, 33
ppci-asm command line option, 34
ppci-build command line option, 33
ppci-c3c command line option, 32
ppci-cc command line option, 38
ppci-hexdump command line option, 51
ppci-java command line option, 49
ppci-ld command line option, 35
ppci-objcopy command line option, 36
ppci-objdump command line option, 37
ppci-ocaml command line option, 48
ppci-opt command line option, 37
ppci-pascal command line option, 40
ppci-pycompile command line option, 41
ppci-readelf command line option, 42
ppci-wabt command line option, 46
ppci-wasm2wat command line option, 45
ppci-wasmcompile command line option, 43
ppci-wat2wasm command line option, 45
ppci-yacc command line option, 44
-version, -V
ppci-archive command line option, 33
ppci-asm command line option, 34
ppci-build command line option, 33
ppci-c3c command line option, 32
ppci-cc command line option, 38
ppci-hexdump command line option, 52
ppci-java command line option, 49
ppci-ld command line option, 35
ppci-objcopy command line option, 36
ppci-objdump command line option, 37
ppci-ocaml command line option, 48
ppci-opt command line option, 37

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

-html-report, 38
-include <file>, 39
-instrument-functions, 39
-ir, 38
-log <log-level>, 38
-machine, -m, 38
-mtune <option>, 38
-output <output-file>, -o <output-file>, 38
-pudb, 38
-pycode, 39
-report, 38
-std {c89,c99}, 39
-super-verbose, 39
-text-report, 38
-trigraphs, 39
-verbose, -v, 38
-version, -V, 38
-wasm, 39
-D <macro>, -define <macro>, 39
-E, 39
-I <dir>, 39
-M, 39
-O {0,1,2,s}, 39
-S, 38
-U <macro>, -undefine <macro>, 39
-c, 39
-g, 38
-h, -help, 38
source, 38
ppci-hexdump command line option
-html-report, 51
-log <log-level>, 51
-pudb, 52
-report, 51
-text-report, 51
-verbose, -v, 51
-version, -V, 52
-width <width>, 52
-h, -help, 51
file, 51
ppci-hexutil command line option
-h, -help, 50
ppci-hexutil-info command line option
-h, -help, 50
hexfile, 50
ppci-hexutil-merge command line option
-h, -help, 51
hexfile1, 51
hexfile2, 51
rhexfile, 51
ppci-hexutil-new command line option
-h, -help, 51
address, 51
datafile, 51
hexfile, 51
ppci-java command line option
-html-report, 49
-log <log-level>, 49
-pudb, 49
-report, 49
-text-report, 49
-verbose, -v, 49
-version, -V, 49
-h, -help, 49
ppci-java-compile command line option
-instrument-functions, 50
-ir, 50
-machine, -m, 50
-mtune <option>, 50
-output <output-file>, -o <output-file>, 49
-pycode, 50
-wasm, 50
-O {0,1,2,s}, 50
-S, 49
-g, 49
-h, -help, 49
java class file, 49
ppci-java-jar command line option
-h, -help, 50
java jar file, 50
ppci-java-javap command line option
-h, -help, 50
java class file, 50
ppci-ld command line option
-entry <entry>, -e <entry>, 36
-html-report, 35
-layout <layout-file>, -L <layout-file>, 35
-library <library-filename>, 35
-log <log-level>, 35
-output <output-file>, -o <output-file>, 35
-pudb, 35
-relocatable, -r, 35
-report, 35
-text-report, 35
-verbose, -v, 35
-version, -V, 35
-g, 35
-h, -help, 35
obj, 35
ppci-objcopy command line option
-html-report, 36
-log <log-level>, 36
-output-format <output_format>, -O <output_format>, 36
-pudb, 36
-report, 36
-segment , -S , 36
-text-report, 36

Q

R

S

T

U

V

W

X